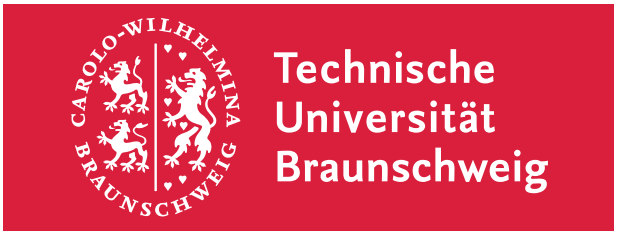
TU Braunschweig



Master's Thesis

Sampling strategies for generating scenarios for simulation-based validation of advanced driver assistance systems

Author:

Lukas Birkemeyer

January 08, 2021

Advisors:

Prof. Dr. Ina Schaefer

Institute of Software Engineering and Automotive Informatics

Prof. Dr.-Ing. Marcus Magnor

Institute of Computer Graphics

https://doi.org/10.24355/dbbs.084-202102191146-0

Birkemeyer, Lukas:
Sampling strategies for generating scenarios for simulation-based validation of advanced driver assistance systems
Master's Thesis, TU Braunschweig, 2021.

https://doi.org/10.24355/dbbs.084-202102191146-0

Abstract

Scenario-based testing is a common approach to verify and validate Advanced Driving Assistance System / Autonomous Driving (ADAS/AD) of motor vehicles. The main challenge in scenario-based testing is the selection of a finite number of scenarios to represent an infinite amount of possible scenarios. Beyond that, there is no metric to evaluate scenarios thus the quality of the testing process.
We introduce a generic process chain to ensure traceability and reproducibility of scenario selection, by generating scenarios automatically. A Feature Model (FM) builds the input data for our process chain. We identify three concepts to represent a scenario using a FM. We create a tool to transfer a configuration of the FM into a concrete scenario. A sample represents a set of scenarios, we define them as scenario suite.
We evaluate the quality of a scenario suite by applying its scenarios to various mutants of driving functions in a simulation tool. The quality of the scenario suites is then determined by the number of discovered mutants. We evaluate the influence of various FMs in combination with common sampling algorithms such as ICPL, Chvatal, and IncLing, using an Autonomous Emergency Braking (AEB) as subject system.
We discover a correlation between FM and mutation score as well as between mutation score and sampling algorithm. Within a scenario suite, we identify a strict separation between scenarios that are good to kill a mutant and those which are not. We discover, that sampling algorithms that aim for feature interaction coverage produce stronger scenario suites than feature-wise sampling algorithms. An evaluation of the relevance of single features on the mutation score provides features that are frequently involved in scenarios that are good to kill mutants. Beyond that, we discover a correlation between scenario suite and mutants that affects the mutant detection.

https://doi.org/10.24355/dbbs.084-202102191146-0

https://doi.org/10.24355/dbbs.084-202102191146-0

Contents

https://doi.org/10.24355/dbbs.084-202102191146-0

https://doi.org/10.24355/dbbs.084-202102191146-0

List of Figures

https://doi.org/10.24355/dbbs.084-202102191146-0

https://doi.org/10.24355/dbbs.084-202102191146-0

List of Tables

https://doi.org/10.24355/dbbs.084-202102191146-0

https://doi.org/10.24355/dbbs.084-202102191146-0

List of Code Listings

https://doi.org/10.24355/dbbs.084-202102191146-0

https://doi.org/10.24355/dbbs.084-202102191146-0

List of Acronyms

ACC	Adaptive Cruise Control
AD	Autonomous Driving
ADAS	Advanced Driving Assistance System
ADAS/AD	Advanced Driving Assistance System / Autonomous Driving
AEB	Autonomous Emergency Braking
AEB VRU	Autonomous Emergency Braking for Vulnerable Road Users
CIT	Combinatorial Interaction Testing
CMIT	CarMaker Interface Toolbox
ECU	Electronic Control Unit
FM	Feature Model
GUI	Graphical User Interface
HiL	Hardware-in-the-Loop
HMI	Human Machine Interface
IDE	Integrated Development Environment
MiL	Model-in-the-Loop
SEC	Safety Envelope Controller
SiL	Software-in-the-Loop
SPL	Software Product Line
SUT	System Under Test
ViL	Vehicle-in-the-Loop
VTD	Virtual Test Drive
VVE	Virtual Vehicle Environment

https://doi.org/10.24355/dbbs.084-202102191146-0

https://doi.org/10.24355/dbbs.084-202102191146-0

1. Introduction

In recent years, Advanced Driving Assistance System (ADAS) became an essential part of modern motor vehicles. They support the driver driving under normal conditions, as well as in critical situations, and thereby, increase the safety and comfort of the road traffic [GGS09]. The part of autonomous functionality in motor vehicles is steadily increasing so that the boundary to Autonomous Driving (AD) is disappearing. To this end, the verification and validation of ADAS/AD will face new challenges. Real-world tests require an immense effort, both in terms of time and money. According to Wachenfeld and Winner [WW16], real-world tests are not practicable as unique tests for release. Suitable methods are needed to reduce the number of real test kilometers required.

Scenario-based testing is a suitable approach for verification and validation of ADAS/AD [UMR+15, Sch17, MBM18]. Menzel et al. [MBM18] define three different abstraction levels of scenarios and arrange them according to the development process of ADAS/AD. At the highest level of abstraction, there are *functional scenarios* that are defined along the concept phase. From these, *logical scenarios* can be derived, which form the second level. They contain all involved components and relations to each other within a certain parameter area. *Concrete scenarios* represent the lowest level of abstraction with defined, concrete values. They form a basis for test cases that are suitable for different stages of test and integration according to the right branch of V-model, particularly for *simulation* and field tests.

Simulation is an established method of testing in the automotive industry [JS16, AWS14]. Software functions are mapped and executed on a computer. This method allows to test at an early stage and, compared to real tests, quickly and cost-effectively. Powerful tools are used to test single components as well as overall vehicles including their environment. Regarding scenario-based testing, a simulation can be understood as test bench, while concrete scenarios build parts of test data and test cases [Boa19]. In combination with the function under test bugs might be found virtually.

However, a challenge of scenario-based testing with regard to verification and validation is the selection of scenarios. The following are only two of still open questions:

https://doi.org/10.24355/dbbs.084-202102191146-0

Which scenarios are relevant? How to ensure that sufficient testing has been carried out? Today, test cases are usually developed by experts based on the specifications of the driving function. The simulation results are influenced by expert knowledge. Furthermore, creating scenarios manually is time and resource intensive. To increase efficiency and objectivity, it is necessary to find a method to create the scenarios or test cases automatically.

Vogelsang et al. [VWR18] understand a simulation tools for ADAS/AD as a configurable system. The basis for this is a FM. The FM logically arranges individual features in a tree structure. The FM represent the scenario space of the simulation tool. With an increasing number of features, this scenario space becomes arbitrarily large. To test an ADAS/AD on all possible scenarios is infeasible. For efficient testing smaller but significant representative subsets of the scenario space need to be selected. These subsets have to be parameterized, translated into the simulation, and executed. According to Vogelsang et al. [VWR18], the selection of features to build a configuration has a significant influence on the results of the simulation. To this end, the impact of the feature selection on the validation of ADAS/AD needs to be examined and evaluated. Further the impact of single features of the FM and their combination needs to be evaluated.

It is necessary to define an objective metric that can be used to evaluate the resulting scenario suites. One approach is to define a *mutation score* determined due to *mutation testing* [Sin11]. Therefore, we specifically seed errors into the model to be tested to create so-called mutants. In order to draw a conclusion to test cases the test environment has to remain the same. In doing so, a *mutation score* that indicates how many mutants are killed by a kill criterion can be used to evaluate scenario suites. Killing a mutant means to identify it as a faulty driving function. The kill criterion is defined according to the specification of the ADAS. The higher the mutation score, the stronger the scenario suite.

Contribution of this thesis

Inspired by Vogelsang et al. [VWR18], we examine the approach of using FMs to generate scenario suites automatically. Later these scenario suites may be useful to validate ADAS/AD. We set up a generic process chain to support research activities within this topic. Setting up this process chain, we examine whether FMs are useful to generate scenario suites. Thereby, we look at different sampling strategies. To compare various scenario suites, we use a mutation score. We examine whether the mutation score is a practical metric to rate scenario suites.

We define the following key questions:

- Are feature models and sampling useful to generate scenario suites automatically?
- Is the mutation score a practical metric to rate scenario suites?

https://doi.org/10.24355/dbbs.084-202102191146-0

Structure of the Thesis

The thesis starts with background information in Chapter 2. Then, we introduce our concept for a generic process chain to generate scenario suites automatically in Chapter 3. Afterward, we present in Chapter 4 tool support to realize the given process chain. Thereby, we discuss existing tools and go into details of their interaction with our implemented components. In Chapter 5, we describe experiments on a concrete subject system and discuss the results. Chapter 6 gives an overview of related work and differentiate it from our work. To close our thesis, we summarize our findings in Chapter 7 and elaborate on possible future research directions in Chapter 8.

Running Example Scenario

Within this thesis, we use a running example to which we refer within the following chapters to clarify our explanations. As an example, we introduce the scenario *simple crossing* where a car drives through a city on a straight road at a speed of 50 km/h (see Figure 1.1). Suddenly, a pedestrian crosses the street right in front of the car. According to this event, the AEB of the car triggers to avoid a collision. This scenario helps to better understand the structure of our FM-design concepts.

Figure 1.1: Running example: Screenshot of CarMaker [IPGa]

https://doi.org/10.24355/dbbs.084-202102191146-0

https://doi.org/10.24355/dbbs.084-202102191146-0

2. Background

In this section, we give some background information for the content of this thesis. We begin to explain FMs, and their structure in Section 2.1. Followed by sampling strategies in Section 2.2. We use the sampling strategies to extract configurations from the FM. In Section 2.3, we explain the idea of mutation testing, the calculation of the mutation score, and the usage of mutation testing within the automotive context. Section 2.4 provides background information about testing within the automotive context.

2.1 Feature Model

Feature modeling is a common approach to describe variability [ABKS13]. The modeling process results in a feature model that can be described graphically as a feature diagram. The feature model, describes the composition of the features f and defines combinatorial connections. Apel et al. [ABKS13] define a feature as a behavior of a system that is visible to the end-user. F describes the set of all features. Depending on the use case, dependencies between features may occur.

Feature diagrams build a graphical description of feature models. Feature diagrams setups the features into a tree structure considering the combinatorial connections. Figure 2.1 presents a feature diagram of a feature model. Each node represents a feature, identifiable by its name. The edges of the tree are tagged using various labels to represent the combinatorial connections. A legend shows the specific meaning of symbols used in the feature diagram. *Cross-tree constraints* express feature dependencies, which can not be modeled as a tree structure.
In the following, we use the terminology *feature model* representing the feature model as well as the feature diagram of the corresponding feature model.

For each feature, there are properties like *mandatory*, *optional*, *abstract*, and *concrete*. Besides, the logical relations between parent feature and its features below may be adjusted using *alternative* and *or* connections. Beyond that, there might be *cross-tree constraints* between arbitrary features. In the following, we explain the meaning of the mentioned properties accroding to Figure 2.1.

https://doi.org/10.24355/dbbs.084-202102191146-0

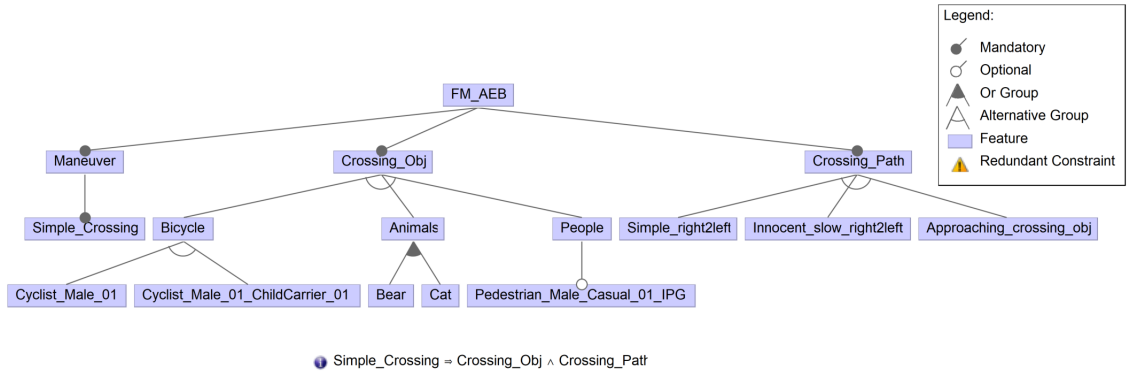


Figure 2.1: Example FM

Mandatory feature: A mandatory feature is a feature that needs to be selected within each configuration where its parent is also selected [ABKS13]. Thus, each mandatory feature is contained in each configuration where its parent feature is also contained. If a feature is mandatory, the feature is marked with a filled bullet on top. All parent features of a mandatory feature are contained in the configuration as well. For instance, the features *Simple_Crossing*, *Crossing_Obj*, and *Crossing_Path* in Figure 2.1 are mandatory features. If a mandatory feature has an optional parent, the mandatory feature only has to be selected, if the parents is selected.

Optional feature: An optional feature is a feature that might be selected optionally for a configuration [ABKS13]. Thus, a configuration may include an optional feature. If a feature is optional, the feature is marked with an unfilled bullet on top. For instance, the feature *Pedestrian_Male_Casual_01_IPG* in Figure 2.1 is an optional feature.

Alternative connection: An alternative connection between two features defines that exactly one child-feature has to be selected [ABKS13]. Thus the alternative connection defines a logical *xor* operator. A configuration may include only one of the child-features. An alternative connection is marked using an unfilled semicircle under the parent feature. In Figure 2.1 we present an alternative connection under the feature *Bicycle*. We may either select *Cyclist_Female_01* or *Cyclist_Male_01_ChildCarrier_01*.

Or connection An or connection between two features defines that at least one child-feature has to be selected [ABKS13]. Thus the or connection defines a logical *or* operator. A configuration has to include at least one of the child-features but may contain more. An or connection is marked using a filled semicircle under the parent feature. In Figure 2.1 we present an or connection under the feature *Animals*. We may select *Bear*, or *Cat*, or *Bear* and *Cat*.

https://doi.org/10.24355/dbbs.084-202102191146-0

Cross-tree constraint: Figure 2.1 contains one cross-tree constraint. The cross-tree constraint is located under the feature diagram in Figure 2.1. A cross-tree constraint uses the implication operator ($\Rightarrow$) in combination with boolean algebra using the feature names to identify features [ABKS13]. Within this example, the cross-tree constraint leads to a redundant definition due to further logical connections between the contained features. The Integrated Development Environment (IDE) FeatureIDE analyses the feature model and marks these issues using a small yellow sign (see legend of Figure 2.1). There are similar red signs that mark dead features. Dead features may not be selected due to the given constraints and logical connections.

2.2 Sampling

Testing of all valid configurations of a system with many variants is infeasible. Combinatorial Interaction Testing (CIT) [CDFP97] is commonly used to reduce the number of tests by sampling a subset of all valid configurations [CDS08, YCP06, CDS07]. According to Kuhn et al. [KWG04], the failure-triggering fault interaction number that triggers failures is quite low. However, the core idea of CIT is to sample, focusing on the combinations of features. This results in a sampling of t-wise combinations. For $t = 1$ the sampling is called feature-wise and for $t = 2$ the sampling is called pair-wise [ABKS13]. Beyond that, CIT provides the possibility to detect faults that are caused due to the interaction of various code segments efficiently.

In contrast to feature-wise sampling, the CIT sampling does not focus on the presence or absence of a single feature, but on the combination of t features. The coverage evaluates the ratio of features for feature-wise sampling or the ratio of combinations for t-wise sampling according to the overall possible number. If all possible features or combinations are included, the coverage is called complete [ABKS13].

There are multiple algorithms to extract a subset of all valid configurations using feature-wise as well as for t-wise sampling [JHF11, JHF12, AHKT+16]. In the following, we present the sampling algorithms Chvatal, ICPL, and IncLing that we use within this thesis.

Chvatal

Chvatal [Chv79] provides an algorithm that calculates a solution for the set-covering problem heuristically. However, the solution is not minimal thus the Chvatal algorithm is greedy.
Johanson et al. [JHF11] adapts the algorithm provided by Chvatal [Chv79] to create configurations of a real FM. Johanson et al. [JHF11] create t-wise samples. In the following we use the terminology Chvatal to describe the Chvatal algorithm [Chv79] adapted by Johanson [JHF11]. Chvatal generates a set U of all valid and invalid t-tuples. Then Chvatal generates a configuration and add combinations of t-features until the configuration becomes invalid. The algorithm removes the last t-tuple that leads to an invalid configuration and add the resulting configuration to the covering set C. Beyond that, Chvatal removes the t-tuples contained in the configuration, from the set U and also all t-tuples that do not lead to a valid configuration, from the set U, if the number of new tuples in the configuration is

https://doi.org/10.24355/dbbs.084-202102191146-0

less then the number of features of the FM. The algorithm repeats all steps while there are t-tuples in the set U.

ICPL

Johanson et al. [JHF12] improve the CHVATAL algorithm to generate configurations for a configurable system [JHF11] that is based on the Chvatal greedy heuristic algorithm [Chv79]. Johanson et al. designed ICPL [JHF12] to handle large FMs. For this purpose, they extend the CHVATAL algorithm by an identification of invalid t-sets within an early stage. Thus they reduce redundant work. Beyond that, they adapt the detection of covered t-sets resulting in a faster and more efficient variant. ICPL consists of various subroutines that are data-parallel thus they can be performed in parallel. A parallelization results in a better performance in comparison with CHVATAL. Johanson et al. [JHF12] compete ICPL to common sampling algorithms. Within the experiments, Johanson et al. [JHF12] identify ICPL as the algorithm with the highest scalability for t-wise sampling using $1 \leq t \leq 3$.

IncLing

INCLING is a pair-wise sampling algorithm provided by Al-Hajjaji et al. [AHKT+16]. The INCLING algorithm generates configurations incrementally. Thus, in contrast to other common sampling algorithms, INCLING provides configurations during the sampling algorithm execution. For this reason, INCLING provides efficient testing due to possibility of parallel sampling and testing.
However, INCLING [AHKT+16] bases on ICPL [JHF12], implementing modifications to improve its performance such as the incremental approach as previously mentioned. Beyond that, INCLING starts removing invalid feature combinations to save time afterward. For the selection of feature pairs, INCLING ranks the feature pairs according to the previously generated configurations. Thus the ranking indicates the probability that the feature pair will be selected within the current configuration. Another major modification of INCLING according to ICPL is the detection of dead or core features. While ICPL evaluates the pair of features simultaneously, INCLING evaluates each feature individually. According to Al-Hajjaji et al. [AHKT+16], this results in an increased overall performance.

2.3 Mutation Testing

Mutation testing is an approach to improve test data that is used to test software components according to DeMillo et al. [DLS78]. The fault-based testing technique evaluates the test set regarding the ability of fault detection [JH10]. Jia and Harman [JH10] provides a survey of mutation testing. The mutation testing technique bases on the coupling effect [DLS78] and competent programmer hypothesis [DLS78].

Using mutation testing so-called mutants $P_M = \{P_{M_1}, P_{M_2}, ..., P_{M_n}\}$ are generated based on the original program P. The mutants are copies of a program inserting mutation operators. It is assumed that the original program from which the mutants are created has no bugs otherwise the original program has to be fixed. Mutation operators are modifications within the program. For example, the mutation operator

https://doi.org/10.24355/dbbs.084-202102191146-0

replaces a logical AND-connection with an OR-connection, or it negates a boolean variable. Thus the mutation operator explicitly insert bugs into the program resulting in mutants. Subsequently, the tests are executed using the original program and each mutant as System Under Test (SUT). The expected output of the test cases commonly refers to the specification of P.
In Figure Figure 2.2, we present the workflow of mutation testing. First, the test cases are executed using the original program P as SUT. If P does not pass the test cases, P needs to be fixed until it does. Finally, the same test cases are executed using each mutant of P_M. The evaluation considers how many mutants are killed using the test set. Killing a mutant means, that the mutant program does not pass the test case. Otherwise, a mutant is called alive or survived. If not all mutants are killed, the mutants in P_M needs to be analyzed. There might be mutants P_{M_x} that are equivalent to P considering their behavior. Grün et al. [GSZ09] evaluate reasons for equivalent mutants. For instance, the mutation operator is implemented in dead code, or the mutation operator improves the original program P in speed. Equivalent mutants can not be detected in the most cases [BA82]. Thus the mutants in P_M needs to be analyzed manually according to equivalent mutants, update P_M, and rerun the test cases. Subsequently, the test set can be improved to kill more mutants.

To evaluate the strength of a test set regarding the ability of fault detection the mutation score (MS) is calculated according to Equation 2.1. Let P_{M_k} be the set of all killed mutants. The mutation score is the ratio of the number of killed mutants divided by the number of overall mutants. We aim for the mutation score $MS = 1$. The higher the mutation score, the stronger the test set.

$$MS = \frac{|\{P_{M_k}\}|}{|\{P_M\}|} \tag{2.1}$$

Jia and Harman [JH10], summarise the problems of mutation testing. They point out, that mutation testing is an effective technique to evaluate the quality of a test set. Nevertheless, mutation testing takes a lot of effort due to the high number of tests that needs to be executed. Each test case has to be applied on each mutant. However, there are investigations to reduce the effort, due to reduce the number of mutants. For instance, Acree [AJ80] suggests to select mutants, and Hussain [Hus08] investigates clustering of mutants.

Altinger et al. [AWS14] investigate testing methods within the automotive context. They identify that mutation testing is not often used neither for research activities nor pre-development or series-development. They map these findings with the fact, that mutation testing is designed for white box testing and the source code is not fully available, especially for series-development. Jia and Harman [JH10] point out that program mutation testing is performed for white-box-testing by inserting bugs into the source code. Besides the program mutation, specification mutation testing is available for black-box-testing. However, Mevenkamp [Mev19] uses mutation testing to evaluate the influence of scenario fuzzing for the testing of autonomous vehicles. Thereby, Mevenkamp uses the tool SIMULTATE [PRWN16] to implement mutation operators into a Simulink model. According to Altinger et al. [AWS14], Simulink is

https://doi.org/10.24355/dbbs.084-202102191146-0

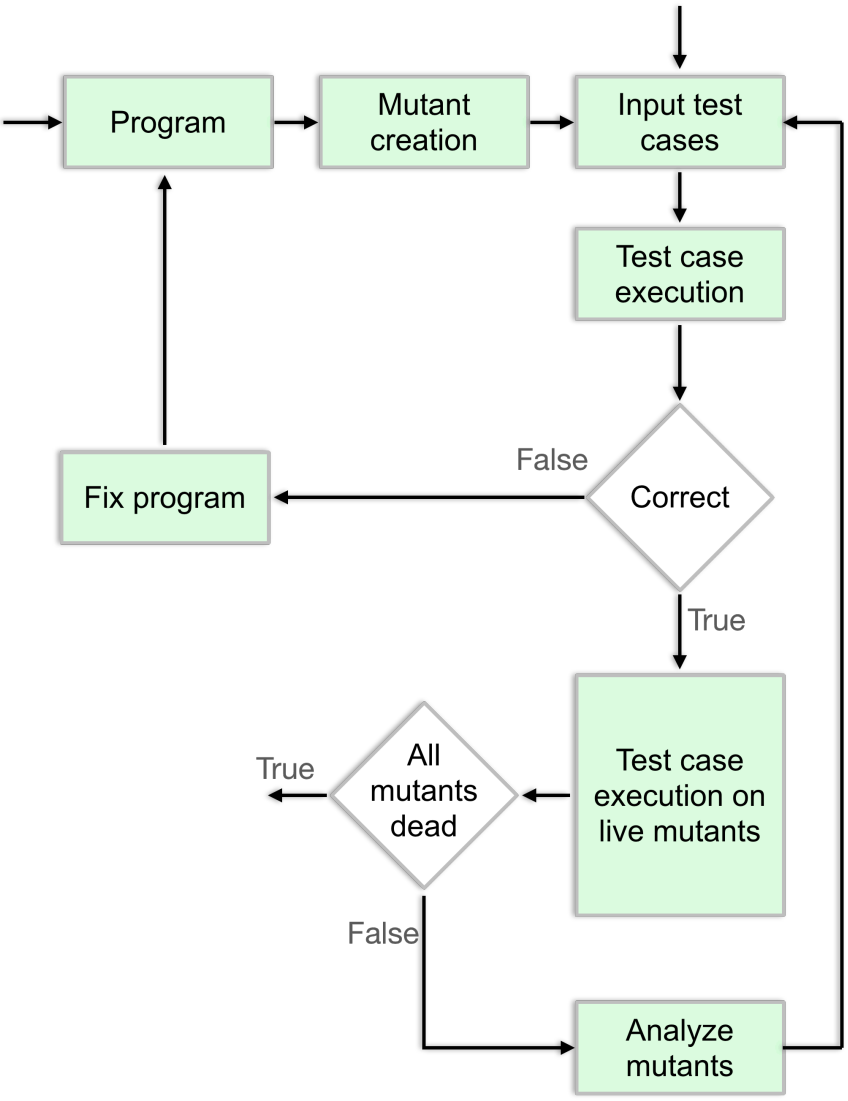


Figure 2.2: Workflow of mutation testing according to [OU01]

https://doi.org/10.24355/dbbs.084-202102191146-0

a tool that is commonly used for model-based source code development within the automotive industry.

2.4 Testing within the Automotive Context

Within this section, we give some background informations regarding the testing within the automotive context. First, we present test concepts that are used. Then we reason the need for scenario-based testing for the validation of AD-functions. Subsequently, we introduce CarMaker, a simulation tool for automotive applications.

Test Concepts within the Automotive Context

Wachenfeld and Winner [WW16] present commonly used testing concepts within the automotive context. Thereby, Wachenfeld and Winner [WW16] highlight, that current testing concepts are designed to test ADAS that mainly support the driver. This comes from an investigation of automation levels for ADAS/AD. Wachenfeld and Winner [WW16] mention the driver as a backup thus a verification and validation has to ensure controllability of the vehicle and the ADAS. The testing process is in accord with the development process [WWP+14]. The development process mainly consists of two sections: Development / Design and Verification / Validation. While, for instance, the design of the test case specification is located in the first section (Development / Design), the test execution is located in the second section (Verification / Validation). According to Wachenfeld and Winner [WW16], the test case generation for ADAS bases on the driver-vehicle system.

However, there are various methods to execute test cases for different abstraction levels within the development process [WWP+14]. Commonly, in-the-Loop methods are used within the automotive context. In the lowest abstraction level, there are Model-in-the-Loop (MiL) and Software-in-the-Loop (SiL). Both methods are designed for a test execution in an early stage using simulation. Thus there is no specific hardware needed to execute the test cases. MiL is designed to test a single model while SiL is designed to test a software component where models are integrated, for instance, a software for an Electronic Control Unit (ECU). At a higher abstraction level, Hardware-in-the-Loop (HiL) and Vehicle-in-the-Loop (ViL) enable to test the software in combination with the hardware for which the software is designed. For instance, HiL enables to test an ECU including the software due to a simulation of the input data for the ECU. ViL [Boc09] combines test of the overall vehicle with a simulation.
All these in-the-loop methods are designed to reduce the number of cost-intensive real test by detecting faults in an early stage. Nevertheless, in-the-loop methods use simulation by simplifying the reality. Thus the vehicle needs to be tested in the real world under real conditions as well [WW16].

Scenario-based Testing

Wachenfeld and Winner [WW16] investigate the amount of needed real test kilometers for an autonomous vehicle theoretically using a statistical approach. They conclude, that the the validation and testing of AD products is an essential challenge thus the release. Schuldt [Sch17] suggests to use scenario-based testing to

https://doi.org/10.24355/dbbs.084-202102191146-0

validate ADAS/AD. Ulbrich et al. [UMR+15] define the termes scene, scenario, and situation within the context of ADAS/AD. Menzel et al. [MBM18] define three different abstraction levels of scenarios in accord with the development process of motor vehicles. *Functional scenarios* semantically represent the entities and relations on the highest abstraction level. For the definition, they use a keyword-based vocabulary. *Logical scenarios* define parameter ranges for the entities used in the functional scenarios. For instance, they define the probability distribution of the road width. *Concrete scenarios* build the lowest abstraction level and define concrete values within the parameter ranges. The concrete scenarios are designed to derive test cases for various testing methods.

Bagschik et al. [BMKM18] present an approach to generate functional scenarios knowledge-based. The generation bases on an ontology resulting in a scenario-graph. The knowledge for the scenario generation is structured within a five-layer model introduced by Schuldt [Sch17] and adapted by Bagschik et al. [BMKM18]. Menzel et al. [MBI+19] introduce an automated transformation of functional scenarios to logical scenarios, based on the functional scenarios generated as defined by Bagschik et al. [BMKM18]. According to Menzel et al. [MBI+19] there is no metric to evaluate the resulting scenarios. Thus, both, Bagschik et al. [BMKM18] and Menzel et al. [MBI+19] investigate the resulting scenarios regarding a correct transformation.

However, there are still open research questions: How valid are all models that are used for simulations? Which scenarios are relevant? How to ensure sufficient testing?

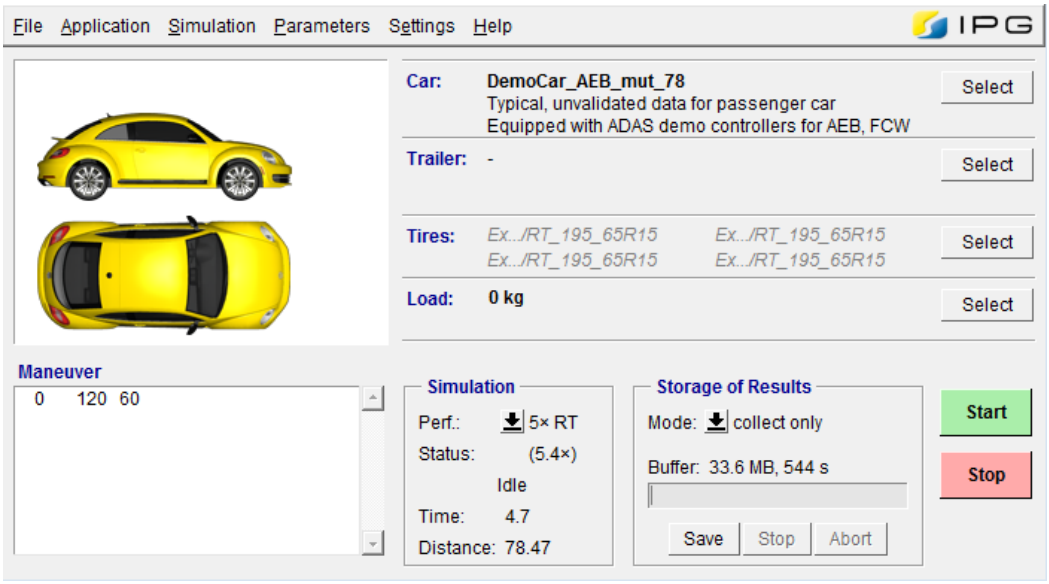
CarMaker

CarMaker [IPGa] is a simulation tool within the automotive context. CarMaker is provided by IPG Automotive GmbH and designed to simulate the Virtual Vehicle Environment (VVE) [IPGd]. The VVE consists of a virtual vehicle, a virtual road, and a virtual driver. The virtual vehicle implements a mathematical model of a road vehicle. For instance, the model implements a chassis, a powertrain, and ADAS. The virtual road includes a model of a road including its specifications such as width and length. The virtual road model also implements environmental informations such as wind, sun position, and obstacles. There are two approaches to implement a virtual driver in CarMaker. First, the user can exactly define the driver's behavior by implementing a driver sequence. Second, the user can use a smart driver that focuses on a given trajectory but may react like a human driver within certain limits. For instance, the driver stops the car if a traffic light is red.

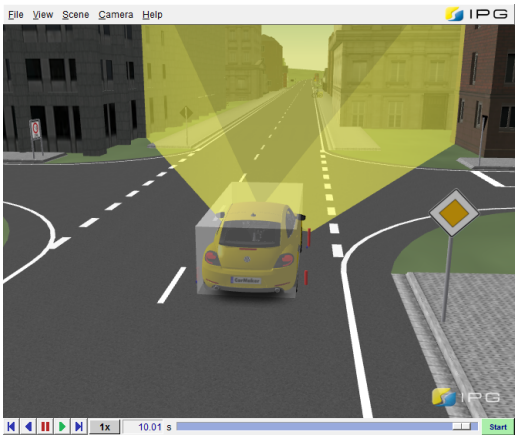
In addition to the VVE, CarMaker provides an CarMaker Interface Toolbox (CMIT) [IPGd]. The CMIT implements various tools to control, parameterize, analyze, and visualize the VVE simulation. The CarMaker GUI is the main component of the CMIT (see Figure 2.3 (a)). Using the CarMaker GUI, the user can, for instance, start or stop the VVE simulation. Beyond that, the CarMaker GUI leads to other tools such as IPGMovie and IPGControl. IPGMovie (see Figure 2.3 (b)) is a tool to visualize the VVE. IPGControl is a tool to plot various inputs and outputs during a simulation. Another component of the CMIT is the TestManager. The TestManager (see Figure 2.3 (c)) is designed to automate testing using CarMaker. The

https://doi.org/10.24355/dbbs.084-202102191146-0

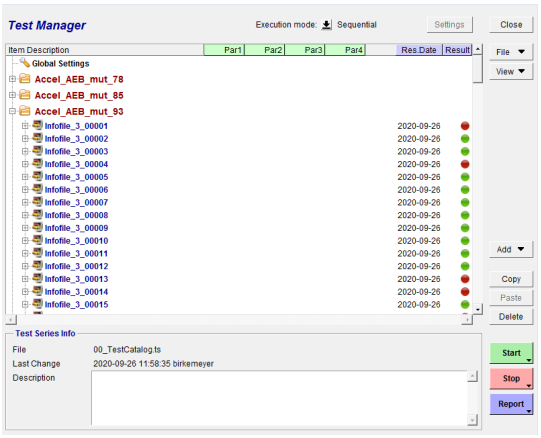
TestManager provides a Graphical User Interface (GUI) as well as a ScriptControl interface.



(a) CarMaker GUI



(b) IPGMovie



(c) TestManager

Figure 2.3: Screenshots of CarMaker Interface Toolbox - Tools of CarMaker [IPGa]

CarMaker parameterizes the VVE using so-called TestRuns. Each TestRun defines a scenario thus, we also call it a scenario or simulation within this thesis. A TestRun is implemented using InfoFiles containing specific syntax according to IPG Automotive GmbH [IPGb, IPGc, IPGd]. The InfoFile-syntax consists of key-value pairs which has no specific order within the InfoFiles. A TestRun, parameterizes the vehicle data, the virtual road, the maneuver, and parameters for the driver. Beyond these mandatory components, a TestRun may contain traffic elements, environmental parameters, or parameters for a trailer. Some parameters like the traffic elements are directly stored within the InfoFile, others such as the vehicle data are linked to another specific files. Beyond the parameterization using InfoFiles, CarMaker provides various interfaces to implement models like ADAS or vehicle models to a simulation. The user may implement the models using C-Code or a Simulink model.

https://doi.org/10.24355/dbbs.084-202102191146-0

https://doi.org/10.24355/dbbs.084-202102191146-0

3. Process Chain

To represent a simulation tool as a configurable system, we set up a generic process chain to support research activities. This chapter describes the developed process chain on a conceptual basis. Figure 3.1 gives an overview of the overall process chain. The following sections provide detailed insights into the components of the overall process chain presented in Figure 3.1.

The process chain is designed to investigate the influence of feature selection on the simulation result. Therefore, the process requires a FM, which represents simulation features and their dependencies, as input. We select a subset of various possible scenarios by using sampling algorithms and call this subset a *scenario suite*. Later the scenarios from the scenario suite are transferred into the simulation tool. We simulate the sampled scenarios in combination with ADAS models and a Safety Envelope Controller (SEC). We evaluate the simulation results using a mutation score. Using this process chain, we are able to investigate the correlations between the sampling method and simulation results.

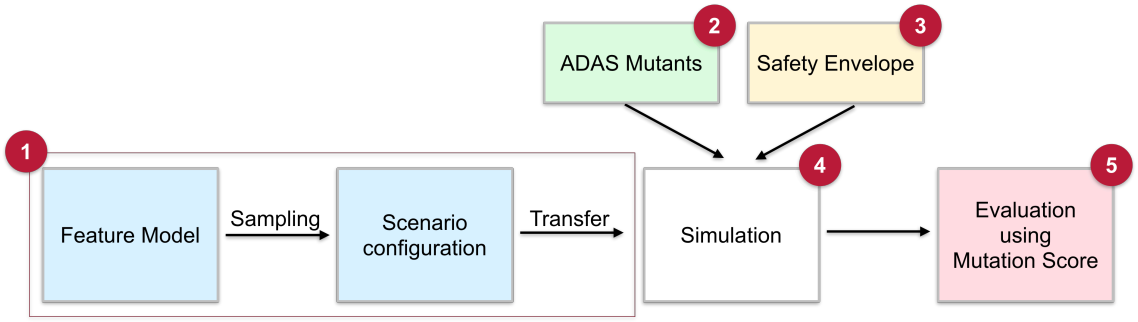


Figure 3.1: Overview of overall process chain

3.1 Feature Model-based Scenario Creation

In this section, we describe how we create scenarios from FMs. Therefore, we structure this section into three parts. Section 3.1.1 gives an insight into the represen-

https://doi.org/10.24355/dbbs.084-202102191146-0

tation of a scenario using a FM. Thereby, we define a classification of features. In Section 3.1.2 we present two core approaches to extract scenarios from FMs. Based on these approaches, we select one example that we use within our thesis. We reason the selection in Section 3.1.3.

3.1.1 Feature Model

The core idea is to model the simulation environment as a FM and to generate scenario suites automatically. The FM builds the input of the process chain and represent the basis for all generated scenarios.
Within this thesis, we use the content definition of a scenario according to Ulbrich et al. [UMR+15]. Considering the sample-based scenario generation using a FM, we define a scenario S to describe a set of features.

$$S = \{f_1, f_2, ..., f_n | \ f \in F\}$$

Beyond that, we define $ScAll$ as a set of all valid Scenarios.

$$ScAll \subseteq \mathcal{P}(F)$$

Definition 3.1. *A **scenario suite** in context of scenario-based testing, describes any combination of scenarios according to Ulbrich et al. [UMR+15]. The scenarios within a scenario suite may match to analyze, verify, or validate a system under test. A scenario suite forms the input data for a test suite in the context of scenario-based testing. A scenario suite ScSu describes a set of valid scenarios.*

$$ScSu = \{S \in ScAll\}$$

A FM is designed to represent a configurable system without considering temporal dependencies. Within the context of scenario-based testing, a single configuration of a FM, represents a scene [UMR+15]. A scenario [UMR+15] is a temporal development of at least one start scene and one end scene. For our simulation we aim for scenario-based testing so that we need to extent the configurations by a development over time. Therefore, we suggest to differentiate all elements of a scenario into the categories *time-invariant* and *time-variant*. We define both terms as follows using the terminology scene and scenario according to Ulbrich et al. [UMR+15]:

Definition 3.2. *A **time-invariant** element in the context of scenario-based testing, describes any element within a scenario that does not change during the scenario. The decisive characteristic is the behavior of the element between two arbitrary scenes within the scenario. A time-invariant element remains in exactly the same state, this does not necessarily except dynamic behavior.*

Definition 3.3. *A **time-variant** element in the context of scenario-based testing, describes any element within a scenario that does change during the scenario. The decisive characteristic is the behavior of the element between two arbitrary scenes within the scenario. A time-variant element varies in its state.*

https://doi.org/10.24355/dbbs.084-202102191146-0

We can classify each element of a scenario either as time-invariant or as time-variant. Commonly, we describe an element using a single time-invariant element or we describe the element using a combination of a time-invariant element and a time-variant element. It is possible to describe, for instance, trajectories using a time-invariant element. If we describe a trajectory as a time-invariant feature we fix the trajectory as a state for all resulting configurations.
Imagine an arbitrary scenario with rain. We can describe rain through the characteristic *rain rate*. If the rain rate does not vary between two arbitrary scenes within the scenario, the state of the rain does not change. Thereby, we classify the element rain as time-invariant within this scenario. Otherwise, if the rain rate varies between two arbitrary scenes, we combine the time-invariant element rain with a time-variant element that describes the delta of the rain rate between scenes.
In a third option, we describe the time-invariant element rain and deposit a rain rate sequence that is fixed for all resulting scenarios.

3.1.2 Scenario Creation

We identify mainly two approaches to include temporal aspects into a FM representation of scenarios. We can either create several scenes and put them together to a single scenario or we can integrate predefined maneuvers as time-invariant features into the FM and change elements individually due to sampling.

Multiple Scene creation

Each scenario bases on a start scene [UMR+15]. After sampling on a FM, we can build these start scenes from the resulting configurations. Based on the start scene we need to generate further scenes that are content-related compatible with the preceding scene. Therefore, we have to look for rules to connect individual scenes. We need to define interfaces to connect all scenes. For example, we can use a similarity-based approach. Pett [Pet18] compares configurations of two samples to determine the difference between two samples. Inspired by this work, we setup up the concept shown in Figure 3.2. There we present four samples containing configurations. The samples are ordered chronologically. Each configuration represents one scene. Sample 0 contains the start scenes. We select one and compare the selected start scene to each configuration of the following sample (Sample 1). For instance, we use the comparison criteria Hamming distance [AHTL+19, Pet18] or Jacard index [Jac12, Pet18]. Based on this comparison we can select the next scenes resulting in a scenario. In Figure 3.2, we mark the dependences between single scenes and highlight the selection. This approach extracts the scenario after sampling using conventional sampling algorithms [JHF12, JHF11, AHKT+16]. In the following, we use the term multiple scene creation using similarity-based sampling to describe this approach.

Another approach of scenario creation using multiple scenes, bases on adaptive sampling algorithms. Figure 3.3 presents an example. There we present four samples containing configurations. The samples are ordered chronologically. Each configuration represents one scene. We sample configurations to generate start scenes (Sample 0) based on a FM, such as we do for multiple scene creation using similarity sampling. In contrast, we do not sample using conventional sampling algorithms

https://doi.org/10.24355/dbbs.084-202102191146-0

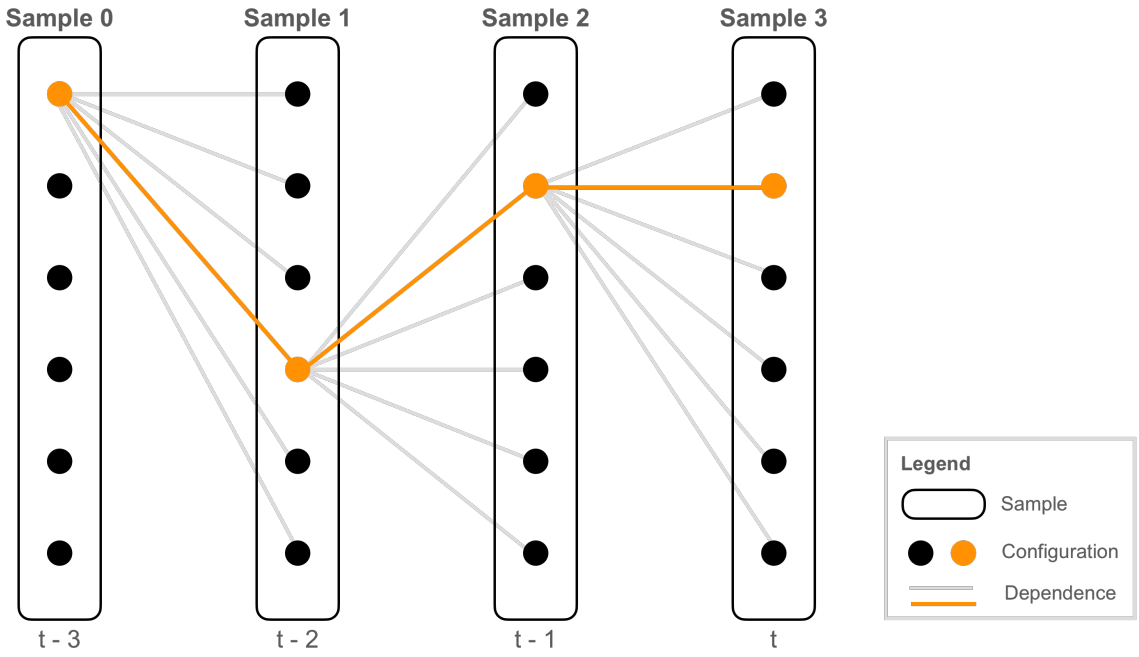


Figure 3.2: Multiple scene creation using similarity sampling

to create the following scenes, but we adapt our sampling algorithms based on the previous scene(s). Therefore, we get a sample (Sample 1) with scenes that validly follow the start scene and select one combination. We mark possible dependences in Figure 3.3 and highlight the selection. Based on these scenes we can generate further scenes resulting in a scenario. A distinction between time-variant and time-invariant elements is necessary. This approach modifies the sampling algorithm rule-based. In the following, we use the term multiple scene creation using rule-based sampling to describe this approach.

Maneuver integration

The approach of maneuver integration into the FM, bases on predefined scenarios. Within these scenarios, we extract time-invariant elements. Beyond that, we identify which elements exist in various scenarios and might be replaced by other elements. Considering our running example: A car drives through a city during the day and a human crosses the street directly in front of the car. Now we can replace the kind of crossing object e.g. with a dog or we add constant rain. Both elements do not change during the scenario. In addition, the elements may vary in another scenario, where the car drives on a winding road and an object stays on the street behind a curve.

In Figure 3.4, we present the composition of a FM using the maneuver integration approach. We define maneuver-features that represent the predefined scenario as a maneuver template. The maneuver template contains all element of the predefined scenario that are time-variant. We fix all elements, trajectories etc. that are contained in the maneuver template thus we get an time-invariant maneuver feature representing the maneuver template. All features that are no maneuver features, are called time-invariant features for this approach.

https://doi.org/10.24355/dbbs.084-202102191146-0

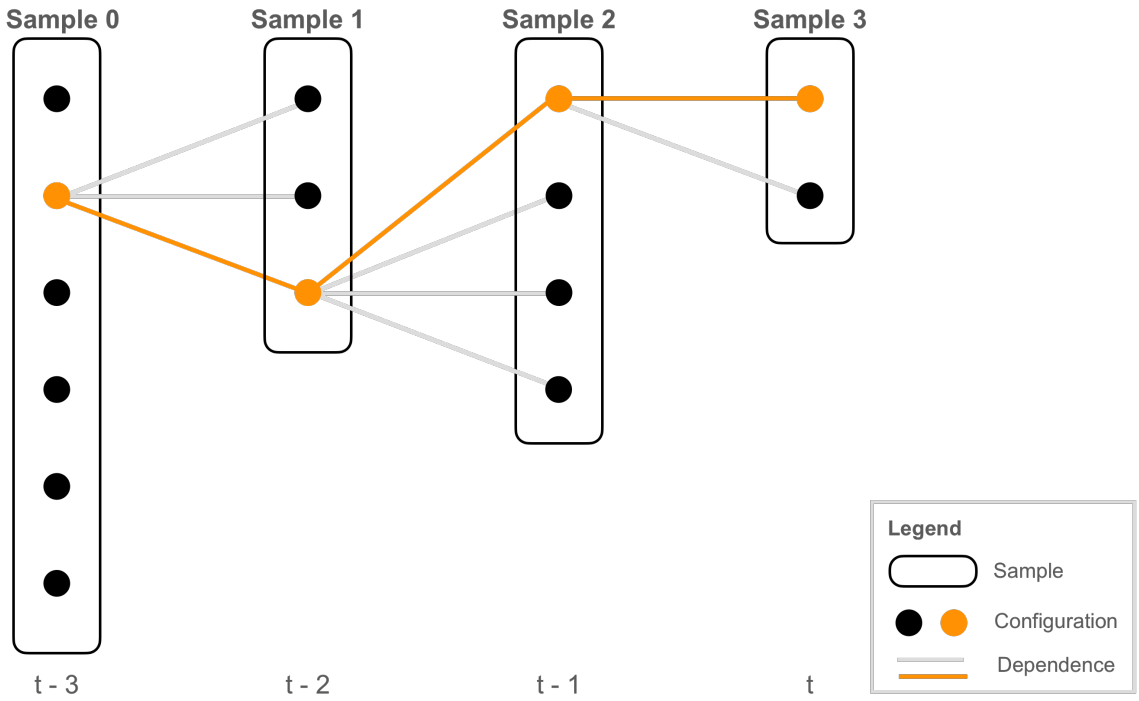


Figure 3.3: Multiple scene creation using rule-based sampling

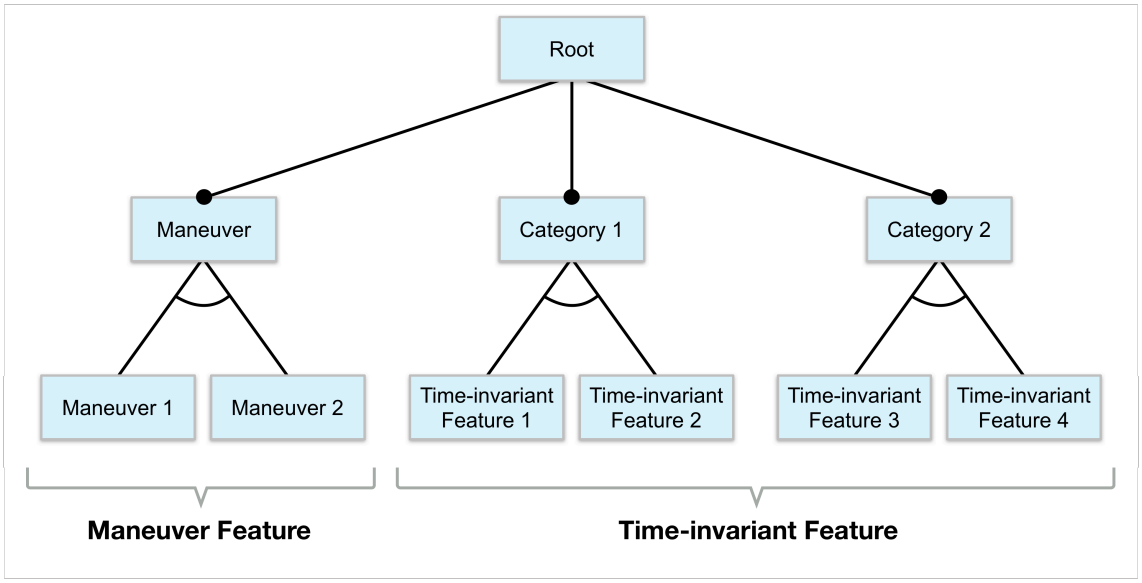


Figure 3.4: Feature composition for maneuver integration approach

https://doi.org/10.24355/dbbs.084-202102191146-0

3.1.3 Concept Design

We aim to implement the process chain as generic as possible, such that we may realize all of the concepts we described within this section. However, we want to evaluate our process chain using a subject system. We aim to investigate the influence of conventional sampling algorithms such as ICPL [JHF12] and Chvatal [JHF11] using our process chain in combination with the subject system. Considering the definition of the concepts we previously mentioned, either the maneuver integration or multiple scene creation using similarity sampling approaches are suitable to investigate common sampling algorithms. The subject system will use the maneuver integration concept due to given example scenarios that are already implemented by IPG Automotive GmbH and used by Mevenkamp [Mev19] for mutation testing in the automotive context. Beyond that, the maneuver-integration concept is more close to the approach of Vogelsang et al. [VWR18] than the multiple scene creation approach using a similarity sampling. Inspired by Vogelsang et al. [VWR18], we build up the FM based on the documentation of a simulation tool. Thereby, we set up the FM in a bottom-up approach, starting with a few features and extend it step by step.

3.2 ADAS and Mutant Generation

In practice, model-based software development is mainly used to implement ADAS especially for rapid prototyping and ECU-programming [JS16, AWS14]. In comparison with conventional code generation, using model based-development the developers do not implement the resulting code by themselves. They rather implement a functional description of the ADAS-specifications as a model and generate the resulting code automatically. An essential advantage of model-based development is the interdisciplinary comprehensibility due to graphical structures such as block diagrams and state machines. Beyond that, the functional description of the ADAS is platform-independent. Therefore, the developed system can simply be transferred to different targets such as different ECUs. Adapted code generators are needed for each kind of target. Another advantage is rapid prototyping [JS16]. Prototypes that we can connect with an existing vehicle are available in the early stages with little effort.

Mutation testing is an approach to evaluate a test set [Sin11, JH10]. For the evaluation of the scenario suites using mutation testing, we need mutants. A mutant is a copy of a program with synthetic modifications. Within our process chain, we identify two approaches to mutate the ADAS model. We can either mutate the model or we can mutate the code after generating code from ADAS model.
Building mutants on the generated code we manipulate the resulting source code [Sin11]. For example, we replace logical operators such as greater than ($>$) with less than ($<$). The mutant generation needs to be platform-specific. Model-based mutation means that we manipulate the ADAS model by inserting mutation operators into the given model inspired by [B+12, PRWN16]. Mutation operators represent well-defined, intended bugs. For example, Figure 3.5 (a) presets an original model for an ADAS, while Figure 3.5 (b) shows a mutant model. We apply mutation operators at the inputs and outputs of the elements in the model. Thereby, we create

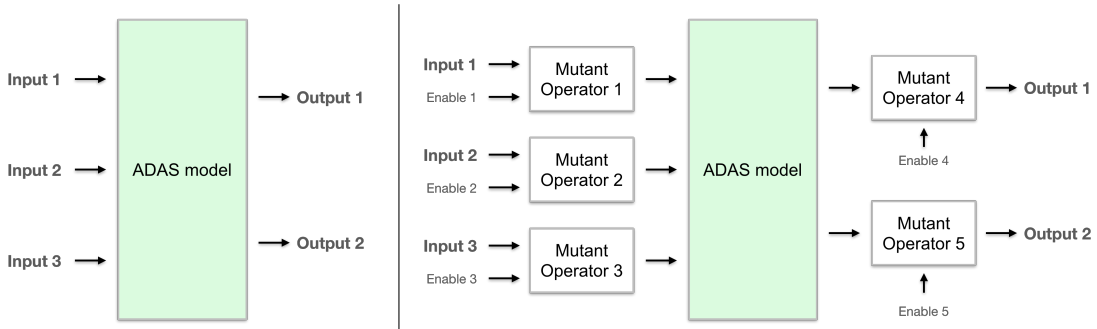
https://doi.org/10.24355/dbbs.084-202102191146-0

a 150% mutant model [SRC+12]. This 150% mutant model contains the original ADAS model extended by mutation operators. We can activate each mutation operator independently due to activation flags resulting in mutants. Subsequently, we can simulate or extract the platform specific source code of the mutants.
Using the 150% mutant model, we can add multiple mutations to a mutant by activating them within the mutant model. This results in a configurable system. We can describe this configurable mutation system using a FM and sample mutant configurations. Thereby, the sampling strategies may influence how hard it is to detect all mutants within a mutant sample. According to this insight, we need to define a specified sampling strategy for mutant creation resulting in a comparable mutation score. In addition to the sampling strategies, the kind of ADAS, implementation, and mutation block integration may also influence the resulting mutants. We aim for a metric that describes how hard it is to detect a single mutant as well as all mutants within a *mutant suite*.

Definition 3.4. *A **mutant test suite** in the context of mutation testing, describes any combination of mutants according to [JH10]. Usually, we create mutant test suites using rule based algorithms to sustain traceability. Each mutant test suite can be described by certain characteristics that match exactly on the combination of contained mutants. A mutant test suite M describes a set of mutants m.*

$$M = \{m_1, m_2, ..., m_n\}$$

However, in practice, one to three mutations are processed [Sin11]. Within this thesis, we aim for first order mutants, according to evaluate whether mutation testing is useful to evaluate scenario suits. First order mutants are mutants that implement one mutation operator.



(a) Original model of ADAS

(b) 150% mutant model of ADAS

Figure 3.5: Structure of mutant model generation

3.3 Safety Envelope Controller

To evaluate the behavior of the ADAS a so-called SEC is useful [Mev19]. This SEC is an ADAS-independent component and evaluates the state of the ADAS. For this purpose, the SEC monitors the input or the output of a SUT and rates, for instance, whether the SUT operates according to its specification or not. Using a SEC to evaluate the behavior of an ADAS, we can calculate independent parameters to observe the SUT. This includes temporal aspects as well as time-invariant parameters.

https://doi.org/10.24355/dbbs.084-202102191146-0

According to evaluate the behavior of an Adaptive Cruise Control (ACC) system, Mevenkamp [Mev19] observes the input of the ACC model and calculates values such as *Time to Collision*, *Time Gap* as well as a *collision indication*. These parameters result in an output of the SEC. The output indicates the state of the ADAS behavior. Using the SEC, Mevenkamp [Mev19] aims to detect defects or undefined behavior of manipulated ACC models. Mevenkamp [Mev19] essentially focuses on vehicle parameters to evaluate safe behavior of the ACC model.

However, within this thesis, we aim to evaluate the behavior of the overall vehicle without focussing on a specific SUT. Reschka [Res16] investigates safety concepts for an autonomous vehicle. Considering the safety of a vehicle, it is relevant to evaluate the current situation the vehicle is located in [Res16, RAAK12, Gey13]. For example, it is very dangerous to stop the vehicle to a standstill on the motorway for no reason. At the same time, it might be necessary to stop the vehicle within or at the end of a traffic jam.

For this purpose, we aim for a SEC that monitors various vehicle parameters as well as its environment. Figure 3.6 presents the input and output structure of a SEC. The goal is to predict whether the overall vehicle operates according to its specification. We define in our thesis, that there must not be a collision with any other object.

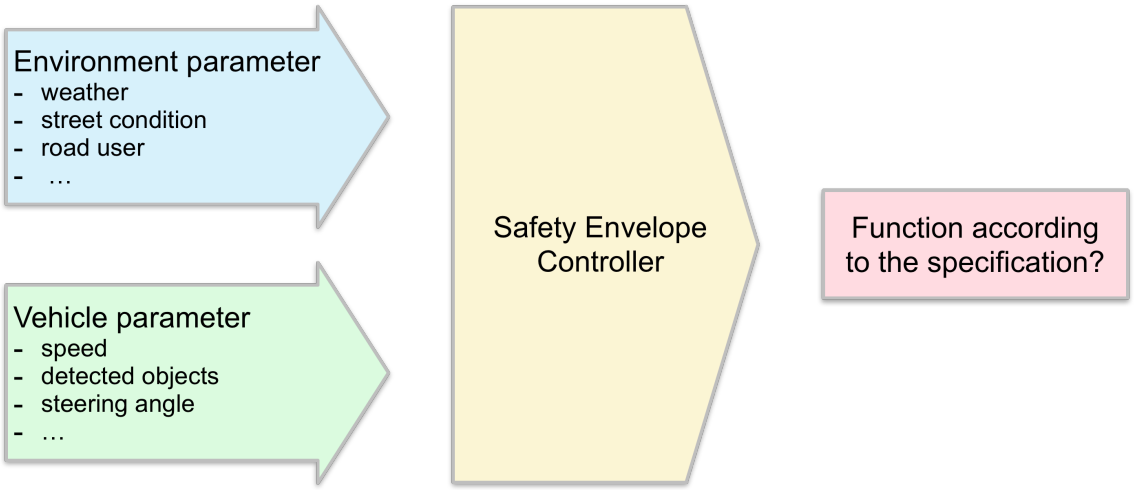


Figure 3.6: Structure of safety envelope controller

Mevenkamp [Mev19] establishes a fault model for mutant behavior (see Figure 3.7). This fault model, differentiates between *defects*, *degradations* and *deviations*. A behavior with *deviations* means, that the behavior of the mutant varies from the original model. *Degradation* builds a subset of *deviation* including mutants with an undefined behavior. The quality of behavior relates to the use case or the subject system. *Defect* builds a subset of *Degradation*. The set *Defect* contains mutants whose behavior results in a failure. Mevenkamp [Mev19] relates this model to autonomous driving and defines a kill criterion that kills mutants whose behavior is located in *degradation* or *defect*. In contrast, we aim to identify whether an ADAS behavior leads to an undefined behavior of the overall vehicle. Thereby, we follow

https://doi.org/10.24355/dbbs.084-202102191146-0

the approach of a SEC, that monitors both, the vehicle parameters and the environment parameters (see Section 3.3). We tolerate ADAS mutant behavior of each set as long the overall vehicle does not leave the defined function according to its specification.

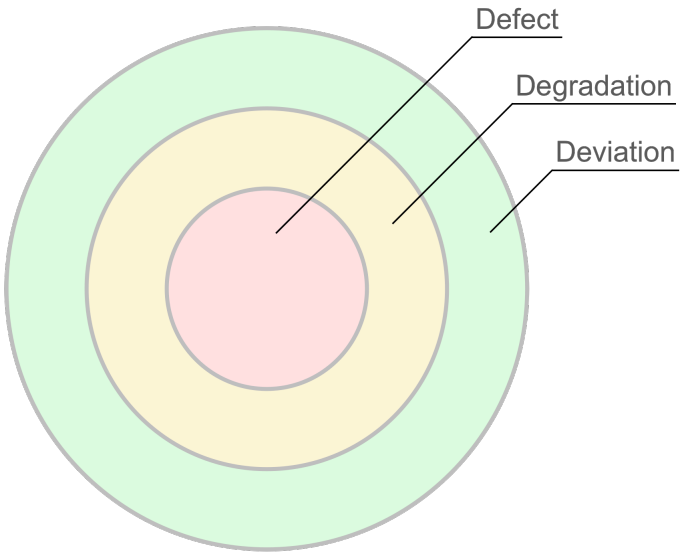


Figure 3.7: Mutant behavior fault model taken from Mevenkamp [Mev19]

3.4 Simulation

There are multiple methods to execute tests. For example, *simulation* is a pure virtual option. Further, there are different approaches to test using various integration environments such as *test benchs* or *lab vehicle* [JS16]. However, we focus on simulation to be hardware independent. In addition, using simulation, we can evaluate the early stages of the development. This early evaluation also maps with our concept of using model-based ADAS, which we presented in Section 3.2. Beyond that, scenario-based testing is mainly suitable for simulation testing [UMR+15].

Input and expected output data are essential elements of a test case [Sin11]. We want to use a simulation tool to execute test cases. The term input data means preconditions (optional) as well as inputs that are necessary during test execution. The input data is applied to the SUT. While testing, we observe the output of the SUT and compare it with the expected output data. If they vary, the test case fails.

In Figure 3.8 we present schematically, how we compose the elements for test case generation. Thereby, we differentiate the structure into *input*, *SUT*, *output*, and *result*. We use scenarios as *input* data and apply these scenarios to the *SUT*. The SUT is represented by ADAS models and mutants of it. The SEC provides the output data that we compare with the expected output data to evaluate the test case. There are two options for a test case result, either a test case fails or passes. We define, that a test case passes if the SUT does not influence the overall vehicle to a change into an undefined behavior, otherwise the test case fails. If a test case passes, the SUT performs so that the overall vehicle operates according to its specification.

https://doi.org/10.24355/dbbs.084-202102191146-0

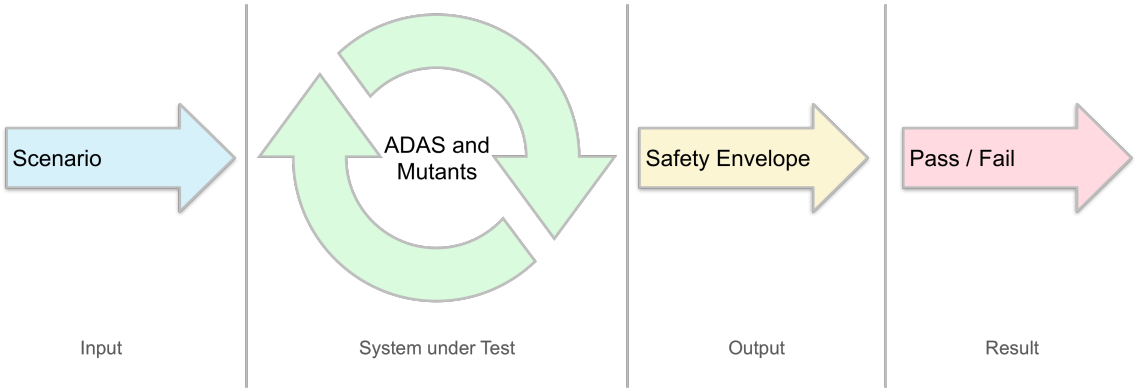


Figure 3.8: Structure of test case elements

According to realize a simulation-based test case execution, we have the following requirements for a simulation tool:
We need to deduce the FM from the simulation tool such that we can use the FM to generate input data for the simulation tool. For this purpose, a comprehensive documentation of the simulation tool is useful to create the FM. We deduce features and their valid combinations from the documentation. Beyond that, the simulation tool has to handle the SUT. Here, we focus on ADAS that are available in a model-based format. Therefore, we need an interface to implement the model-based ADAS into the simulation tool. In parallel, the simulation tool needs to allow monitoring of various parameters, at least the output data of the SEC, to evaluate the test cases. We also require an integrated test case management for efficient testing.

3.5 Evaluation using Mutation Score

We aim to evaluate various scenarios suites. Therefore, we use a mutation score as a metric to evaluate different sampling strategies that we use the generate the scenario suites. Performing mutation testing, we apply a given scenario suite on the origin ADAS. Beyond that, we apply the same scenario suite to each mutant of the ADAS.

Considering mutation testing, we need to define a kill criterion. According to the kill criterion, we classify a mutant as killed or survived. The kill criterion is commonly derived the functional specification of the SUT [JH10]. According to the kill criterion we classify a mutant as *killed* or *survived*. Considering a kill criterion derived from the specifications, the expectation is that the original function passes all test cases and a mutant is killed if it fails. In contrast, within this thesis, we use the kill criterion to define the expected output of the test case. Thus the kill criterion leads to positive or negative test cases.

We define the SEC we presented in Section 3.3 as a kill criterion. The SEC is designed to evaluate whether the overall vehicle operates according to its specification. Varying a single ADAS, we investigate the impact of this ADAS on the behavior of the overall vehicle. There might be correlations with other ADAS. However, we aim to create scenario suites regardless of a specific SUT. We apply the scenario suite

https://doi.org/10.24355/dbbs.084-202102191146-0

to the original SUT and use the SEC to define the expected output. Subsequently, we apply the test cases on the mutants as we described in Section 3.4. We evaluate the outputs of the SEC for each mutant regarding the expected output. We kill a mutant if it does not pass the test cases. Otherwise, the mutant survive.

Based on the information whether a mutant is killed or alive we calculate the mutation score (MS). To that, we use Equation 3.1 inspired by Singh [Sin11]. We count the number mutants of a mutant test suite that we kill using a certain scenario suite. Subsequently, we divide the number of killed mutants by the overall number of mutants. Thereby, we refer to the ground truth number of synthetically generated mutants. The quotient is the mutation score. We do not analyze the mutants according to equivalence. According to Grün et al. [GSZ09], the investigation of equivalent mutants needs takes a lot of effort. Thus we expect a minor mutation score in contrast to mutant equivalence analysis.

$$MS(M, ScSu) = \frac{|\{m \in M \mid m \textit{ killed by some } S \textit{ in } ScSu\}|}{|\{M\}|} \tag{3.1}$$

https://doi.org/10.24355/dbbs.084-202102191146-0

https://doi.org/10.24355/dbbs.084-202102191146-0

4. Tool Support

As a proof of concept, we implement the process chain that we presented in Chapter 3. We describe the tool support and the interaction with implemented tools in this chapter. Thereby, we refer to the structure of Figure 3.1. In Section 4.1.1, we present the scenario creation separated into the design of the FM design and the transformation process. In Section 4.2, we explain the workflow that we follow to generate mutants. Thereby, we present the generation of a mutant model, the integration of ADAS models into the simulation environment as well as the generation of mutant test suites and their integration into the simulation environment. Subsequently, we present the implementation of our SEC in Section 4.3 and the simulation environment as well as the test automation in Section 4.4. Finally, in Section 4.5, we present the workflow for an evaluation of the simulation results using the mutation score.

4.1 Feature Model-based Scenario Creation

This section describes the sub-process where we extract executable scenarios from a FM. Figure 4.1 gives a specific insight into the sub-process chain. We focus on the maneuver integration approach according to Section 3.1.2. Thereby, we extract valid configurations of the FM using various sampling strategies. Each configuration represents one scenario. Subsequently, we transfer each configuration into an executable scenario for the simulation tool.
We structure this section into a description of our FM (Section 4.1.1) and the transfer from configuration to the simulation tool (Section 4.1.2).

4.1.1 Feature Model

There are several tools to implement FMs [ABKS13] [BPPK09]. For example, there is FeatureIDE [TKB+14]. FeatureIDE is an open-source solution to design FMs. Another open-source solution is Captain Feature [Cap]. DarwinSPL is a tool to implement evolving FMs [NES17]. EvoSPL [BPPK09] is another framework that deals with evolving FMs. EvoSPL is desigend to implement EvoFMs [BPPK09].

https://doi.org/10.24355/dbbs.084-202102191146-0

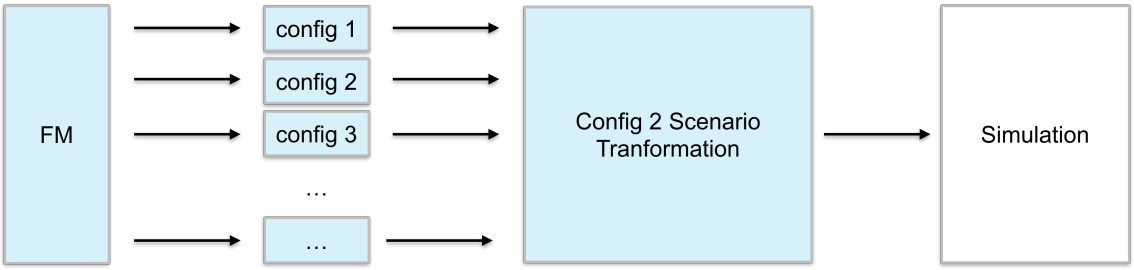


Figure 4.1: Transformation process scenario suites to simulation data (overview)

Beyond that, pure-systems GmbH provides the commercial solution pure::variants to support development processes using software product lines. This includes an implementation of a FM [pur]. The following paragraph provides our intention to use the existing tool FeatureIDE.

FeatureIDE

Within this thesis, we use FeatureIDE [TKB+14], a state of the art framework for feature modeling. Utilizing this open-source framework, the user can graphically implement FMs. According to Apel et al. [ABKS13], FeatureIDE is a tool suitable for academic research and provides integrations for further research tools. We do not need these integrations within our thesis, but we have the possibility for further research activities, anyway. FeatureIDE bases on Eclipse, thus the plug-in mechanism of Eclipse is available as well.

Beyond that, we can use FeatureIDE for testing [AHMK+16]. FeatureIDE provides a product generator. We use the product generator to sample configurations from the FM. FeatureIDE implements the t-wise sampling algorithms CASA, Chvatal, ICPL, and IncLing as well as the possibility to sample all valid configurations and random configurations. The user can select the sampling algorithm and if needed the value of t for t-wise sampling algorithms. Beyond that, FeatureIDE provides the possibility to define a maximum number of configurations. Subsequently, the user can export each configuration within a configuration-file. Each configuration-file contains the names of selected features of the related configuration.

Feature Model Design

As previously mentioned, we implement the FM using the maneuver integration approach. In Figure 4.2 we present an excerpt of our FM related to our running example.

We start to build the FM by considering a concrete scenario such as our running example *simple crossing*. From this scenario, we extract time-invariant elements. At this point, we create a FM that contains two features. The feature named *maneuver* is parent of the feature *simple crossing*.
In a next step we identify time-variant and time-invariant elements. For instance, we identify the type of the crossing object as time-invariant. The object moves between two arbitrary scenes during the scenario but does not change its type. Therefore, we

https://doi.org/10.24355/dbbs.084-202102191146-0

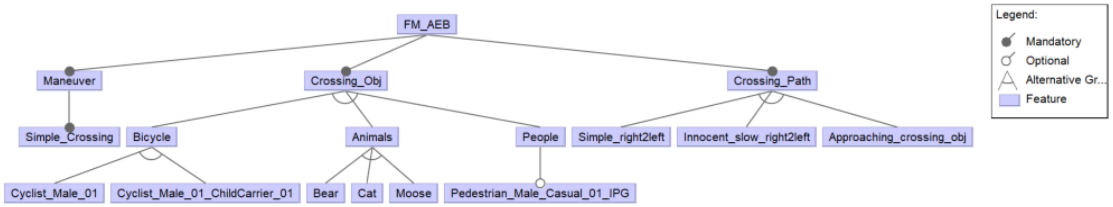


Figure 4.2: FM according to the running example

add the feature *Crossing Obj* to the FM representing the type of the crossing object. Below this feature, we add multiple types of crossing object features. We deduce further possibilities from the simulation tool, referring to the documentation [IPGd, IPGc, IPGb]. For our running example, we take CarMaker of IPG Automotive GmbH as our simulation tool. IPG Automotive GmbH provides some example data with their simulation tool. For instance, we identify different pedestrians, cyclists, and animals. We add these types to our FM (see Figure 4.2). For each simple crossing scenario, we need to define exactly one type of crossing object thus we declare the *Crossing Obj* feature as mandatory. Beyond that, we realize the connection to the features below using alternative constraints. We fix the trajectory of the crossing object as a time-invariant element for all resulting configurations. Within this example, we define three individual trajectories. We define the feature *crossing path* to represent the trajectories. We capture each trajectory as a single feature below the feature *crossing path*. Again we need exactly one trajectory for the simple crossing scenario such that we define the feature crossing path as mandatory and connect the features below using an alternative constraint.

Sampling

We perform sampling, based on the example FM we present in Figure 4.2. Each configuration in a sample corresponds to one scenario due to the maneuver integration concept. The sample builds the scenario suite. We use common sampling algorithms such as ICPL [JHF12], Chvatal [JHF11], and IncLing [AHKT+16] for sampling. These three sampling algorithms are available in the FeatureIDE product generator. Each sampling algorithm extracts a set of configurations that aims to represent the whole configuration space. The sampling algorithms focus on different strategies to reach the coverage, resulting in various sets.
After sampling, each configuration is provided by FeatureIDE as a list of the selected features. FeatureIDE exports each list within a plaintext configuration-file. These configuration-files contain the names of the selected features.
For example, we have a configuration that contains the features *Simple Crossing*, *Cat*, and *Simple right2left*. This configuration represents a scenario where the car drives through a city *(Simple Crossing)* and a cat *(Cat)* crosses the street starting at the right side over to the left side *(Simple right2left)*.

4.1.2 Transfer from Configuration to Simulation

A manual transfer of configuration-files into executable simulation scenarios for a simulation tool is infeasible, due to the large number of configurations. We need

https://doi.org/10.24355/dbbs.084-202102191146-0

a tool to perform this transformation. There are two approaches: The tool has to control the simulation tool during the simulation (online) or we have to manipulate the input data before the simulation starts (offline). An online manipulation of the data means the highest flexibility during the simulation process. The simulation tool CarMaker, for instance, provides an interface to connect with Simulink [IPGb]. Simulink[1] is an IDE for model-based development and bases on MATLAB by The MathWorks, Inc. The automotive commonly uses MATLAB/Simulink for model-based development [AWS14]. However, we can control CarMaker from Simulink so that we can directly simulate the model that is implemented in Simulink. Using an offline manipulation of the input data, the simulation and the data manipulation are separate. According to IPG Automotive GmbH, we expect a better performance of CarMaker if we do not use the CarMaker for Simulink Interface [IPGb]. Finally, we focus on the offline manipulation of the input data, so that we realize an independent simulation process expecting better performance.

Scenario representation

There are various data formats to provide the input data for simulation tools [IPGd, VIRa]. The syntax is, essentially, pretended by the simulation tool. The simulation tool CarMaker, for instance, expect InfoFiles using a key-value pair syntax for a scenario definition [IPGb]. ASAM e.V. recently releases the three specifications OpenCRG [ASAb], OpenDRIVE [ASAc], and OpenSCENARIO [ASAa] to describe the VVE. These standards provide compatibility for different simulation tools. However, not all simulation tools completely support these standards yet. We aim for a generic implementation of our process-chain. Thereby, we create a database structure that we can adapt to various simulation tools. Within this thesis, we focus on the implementation of CarMaker InfoFiles due to a provided workaround using CarMaker.

We explain our implementation focussing on CarMaker specific InfoFile creation. The content of InfoFiles consists of key-value pairs, so that they can be mapped with little effort using the features of FM. We use a template of an InfoFile to build the main structure. These template-files represent the core maneuver but are no executable InfoFiles. In the following, we use the term maneuver template to describe these templates. Listing 4.1 presents a maneuver template for our running example. The maneuver mainly contains time-invariant and time-variant elements of the scenario. For example, there are the basic roadmap and trajectories of certain traffic elements. The template also contains self-defined keys. These keys are designed to attach or replace code at defined positions. Each key starts and ends with a $-sign thus we can detect them easily. The surrounding $-sign is not part of the InfoFile-syntax.

[1] https://de.mathworks.com/products/simulink.html

https://doi.org/10.24355/dbbs.084-202102191146-0

```
1              ...
2              DrivMan.Init.Velocity = $Speed_vut=60
3              DrivMan.Init.GearNo = 4
4              DrivMan.Init.SteerAng = -35
5              DrivMan.Init.LaneOffset = 0
6              DrivMan.Init.OperatorActive = 1
7              DrivMan.Init.OperatorState = drive
8              DrivMan.VhclOperator.Kind = IPGOperator 1
9              DrivMan.nDMan = 1
10             DrivMan.0.TimeLimit = 150
11             DrivMan.0.EndCondition = SafetyEnvelopeCtrl.OverallOK == 0
12             DrivMan.0.LongDyn = Driver 1 0 $Speed_vut=60
13             DrivMan.0.LatDyn = Driver 0
14             Traffic.IFF.FName =
15             Traffic.IFF.Time.Name =
16             Traffic.N = 46
17             Traffic.SpeedUnit = kmh
18             Traffic.0.ObjectKind = Movable
19             Traffic.0.ObjectClass = $Crossing_Object_Class$
20             Traffic.0.Name = Cross_ob
21             Traffic.0.Info = $Crossing_Object_Class$
22             Traffic.0.Movie.Geometry = $Crossing_Object$
23             ...
```

Listing 4.1: Extract of the simple crossing maneuver template

Structure of transformation process

As we said, we use a database structure to implement all simulation specific code elements. We present the structure of the databases for InfoFile creation in Figure 4.3. The structure bases on the structure we presented in Figure 4.1. We expand the existing process by the databases and defined output files. We use the transformation tool (*Config 2 Scenario Transformation*) to generate InfoFiles, as well as a TestSeries-file (*TestCatalog.ts*). We need the TestSeries-file for an automated test case execution in CarMaker. However, we use four different databases that provide data for the *Config 2 Scenario Transformation*. The database *Database_infofile.csv* contains all features that are implemented in the related FM. In this database, we store the code for the InfoFiles, linked to each feature. For maneuver features, this database includes the storage location of the maneuver template. Beyond that, we use a separate database *Database_metadata.csv* for metadata of the InfoFilesand the database *Database_mutant.csv* contains information about the mutants we use for the mutation testing. We store information that we need to generate the TestSeries-file for a test automation in the database *Database_testCatalog.csv*.
In the following, we present the databases in more detail.

Database_infofile.csv In Table 4.1 we present an excerpt of *database_infofile.csv*. We identify a feature according to its name. There is code and a section linked to each feature. We use the code for the generation of InfoFiles. The section defines how to handle the code that is linked to the feature.

https://doi.org/10.24355/dbbs.084-202102191146-0

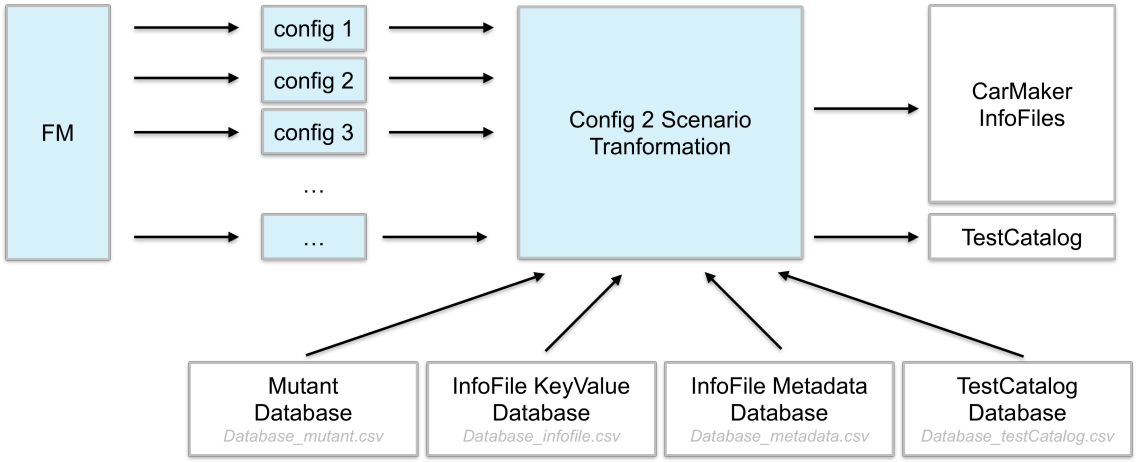


Figure 4.3: Transformation process scenario suites to simulation data (detail)

Feature	**Section**	**Code**	
Simple_Crossing	Maneuver	Simple_Crossing.txt	
Cat	Crossing_Object	3D/Animals/Cat.manim	
fog_50m	Default	Env.VisRangeInFog = 50.0	

Table 4.1: Excerpt of *Database_infofile.csv*

Database_metadata.csv In the *Database_metadata.csv* we store metadata that we add to the InfoFiles. For instance, the metadata contains the mandatory line in InfoFile has to start with. Beyond that, the *Database_metadata.csv* contains when the data was last changed and the CarMaker version for which the InfoFile is designed.

Database_mutant.csv In Table 4.2 we present an excerpt of *database_mutant.csv*. We identify a mutant according to its name. We link each mutant with a Mutation Block. The Mutation Block defines which mutation operator of the 150% mutant model is enabled in the mutant. Beyond that, we link each mutant model with code. We use the code in the InfoFiles to link a mutant model and a scenario.

Mutant_Name	**Code**	**Mutation Block**	
Accel_AEB_origin	Vehicle=DemoCar_AEB_origin	origin	
Accel_AEB_mut_1	Vehicle=DemoCar_AEB_mut_1	[...]/Divide_mut_abs_en	
Accel_AEB_mut_2	Vehicle=DemoCar_AEB_mut_2	[...]/Divide_mut_abs_en1	

Table 4.2: Excerpt of *Database_mutant.csv*

https://doi.org/10.24355/dbbs.084-202102191146-0

Database_testCatalog.csv In Table 4.3 we present an excerpt of *database_testCatalog.csv*. We identify the TestSeries-elements by their name. We link the TestSeries-elements to a section and code. The section indicates how to handle the code. We use the code to generate TestSeries-files. The code contains keys, that are surrounded by $-signs. We replace the keys during the transformation process.

Name	Section	Code	
testcatalog_config	testcatalog	Step.$cntMut$.$cntRun$ = TestRun [...]	
testcatalog_group	testcatalog	[...] Step.$cntMut$.Name = $NameMut$	
test_criterion_bad	criterion	[get SafetyEnvelopeCtrl.OverallOK] == 0	

Table 4.3: Excerpt of *Database_testCatalog.csv*

Transformation Tool

In the following paragraphs, we explain how we implement the transformation from config-files to CarMaker InfoFiles. We implement the transformation tool using Java. Java is a platform-independent programming language and provides both, object-oriented programming and prepared elements to realize a GUI [Abt13]. The transformation tool is not a parser due to multiple input data that we have to connect logically. Thereby, we aim for object-oriented programming, so we can create internal structures. Beyond that, we aim for a guided control for the user, such that we use a GUI as our Human Machine Interface (HMI). The procedure of transformation mainly consists of five steps: *Load Databases*; *Load Configuration-Files*; *Selection of Output Location*; *Transformation*; *TestSeries-File Generation*

Step 1: Load Databases In the first step, *Load Databases*, we ask the operator to select the location of the databases using a JChooser GUI element. We expect all databases within the same folder. We read each database line by line and convert the content of *Database_infofile.csv* into FeatureCode objects. Each object contains all pieces of information that are linked to a single feature in the database. For example, we store the feature name, section, and the InfoFile syntax code. We collect all FeatureCode elements within a hashmap using the feature name as a key. We choose a map for the clear allocation of code and configuration-files. We set up the collection once and use it for each configuration-file therefore, we use a hashmap. Hashing is designed for quick access [Hei11].

Step 2: Load Configuration-Files In the second step, *Load Configuration-Files*, the tool asks the user to choose the directory, where the configuration-files are located. FeatureIDE exports the configuration-files such as we present in Figure 4.4. FeatureIDE numbers the configuration-files consecutively, starting with *00001*. In some cases, numbers are skipped in the numbering of the configurations. This behavior usually appears using t-wise sampling for $t \geq 2$. Thereby, we need to rename and

https://doi.org/10.24355/dbbs.084-202102191146-0

reposition the configuration-files. We perform this by a python-script before starting the Java tool. An integration of this python script into the java tool is conceivable. The Java tool recursively works through the repositioned subdirectories. Thereby we save the exact location of each configuration-file within an ArrayList.

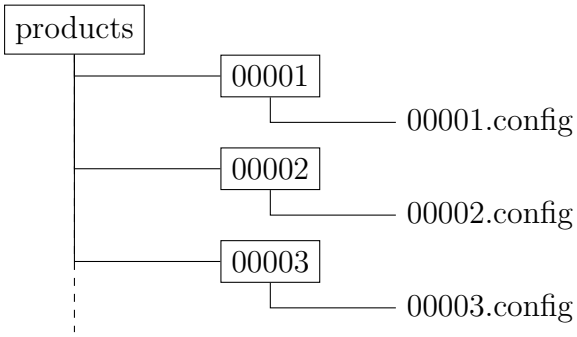


Figure 4.4: Folder structure of configuration files

Step 3: Selection of Output Location Before the Java tool starts the InfoFile creation, the user has to define a location where the Java tool has to store the resulting InfoFiles. The destination folder has to be empty. If it's not, the tool asks the user to choose another folder or to override the content.

Step 4: Transformation We separate the Transformation into the Input workflow and the InfoFile generation. In Figure 4.5, we present the input-workflow of the transformation from left to right. The Java tool reads one configuration-file. We collect all features of the configuration-file and create a set containing all these features. Then, we sort the features according to three categories. The categories result from section entry within the *Database_infofile.csv* database. We differentiate between section *maneuver*, *default*, and any other section named *relevant section key* set.

Figure 4.6 presents the internal structure that we use for the InfoFile generation. We create two lists to buffer data: *File Content List* and *Maneuver List* We present them in the middle of Figure 4.6. To the left of the lists, we describe the relevant input from the databases and to the right the related feature sets from the Input workflow. We start to create the list, *file content*, that we use to buffer the content for the InfoFile. First, we add metadata from the *Database_metadata.csv*. Second, we add information about the SUT from *Database_mutant.csv* to this list. Third, we add the maneuver to the *File Content* list. There is exactly one feature that describes a maneuver due to the rules we implement into the FM. We extract the storage location of the linked maneuver template from the *Database_infofile.csv* and read this maneuver template line by line into a *Maneuver Content* list. Again, the maneuver template contains keys surrounded by $-signs. We check the *Maneuver Content* list for these keys. Concurrently, we map the keys from the *Maneuver Content* with the features in the *section key* set. If a key and a feature match, we replace the key in the *Maneuver Content* through the code that is linked to the feature in the *Database_infofile.csv*. We ensure that all keys are replaced, due to

https://doi.org/10.24355/dbbs.084-202102191146-0

the FM design. Subsequently, we add the content of the *Maneuver Content* list to the *File Content* list. Fourth, we add code that is linked with the features of the default set to the *File Content* list. The corresponding code is stored in the *Database_infofile.csv*. Finally, we transfer the content of the *File Content* list to an empty file resulting in an InfoFile. We repeat the whole transformation process for each configuration-file.

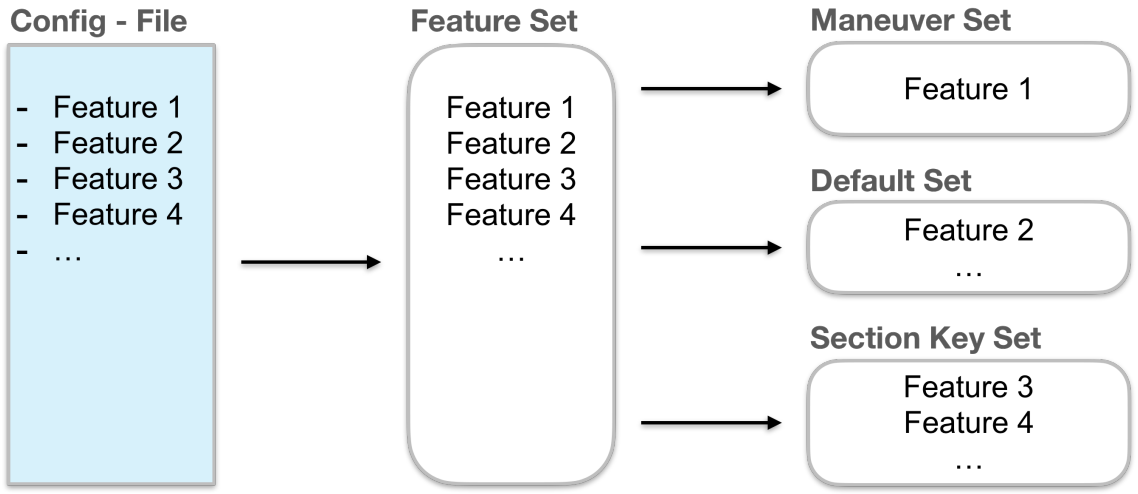


Figure 4.5: Transformation tool: Input workflow

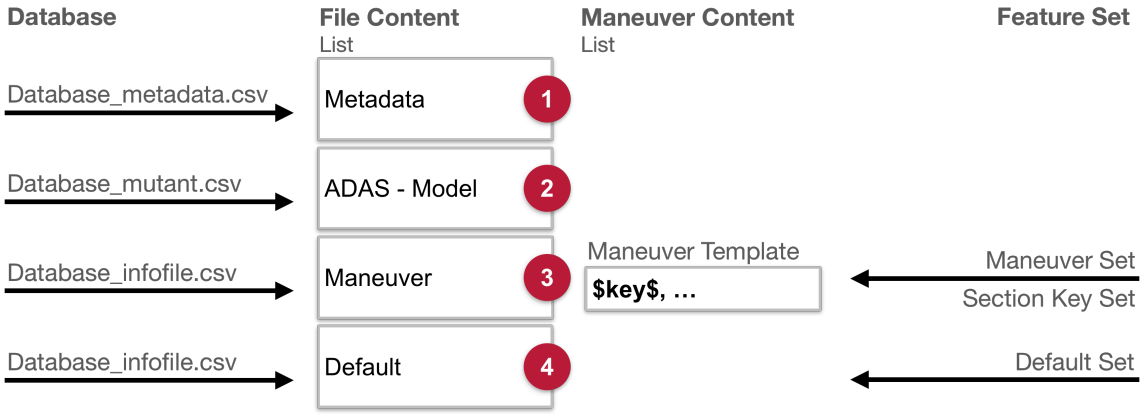


Figure 4.6: Transformation tool: InfoFile generation

Step 5: TestSeries-File Generation Beyond the InfoFiles, the transformation tool generates the TestSeries-file (*TestCatalog.ts*). We use the TestSeries-file in combination with the TestManager to execute the simulation automatically. The TestSeries-file defines the execution order of the simulations and also specifies the expected test case result for each simulation. The code for the TestSeries-file follows the CarMaker InfoFile syntax and a code fragments are stored within the *Database_testCatalog.csv*. Within these code fragments, we implement keys surrounded by the $-sign as we do for the maneuver template. The transformation tool

https://doi.org/10.24355/dbbs.084-202102191146-0

replaces the keys in the code fragments, for instance, by the name of the current InfoFile and stores the resulting code in an empty file.

4.2 ADAS and Mutant Generation

Within this section, we describe our workflow to generate mutants. We present the workflow in Figure 4.7 from left to right. An ADAS model builds the input data. We create a 150% mutant model of the ADAS by using SIMULTATE [PRWN16] and parts of MTAF [Mev19]. Based on this 150% mutant model, we can generate multiple mutants using the script *mutate.py*. We use the Simulink Coder Interface of CarMaker to integrate the mutant test suite into the simulation environment CarMaker.

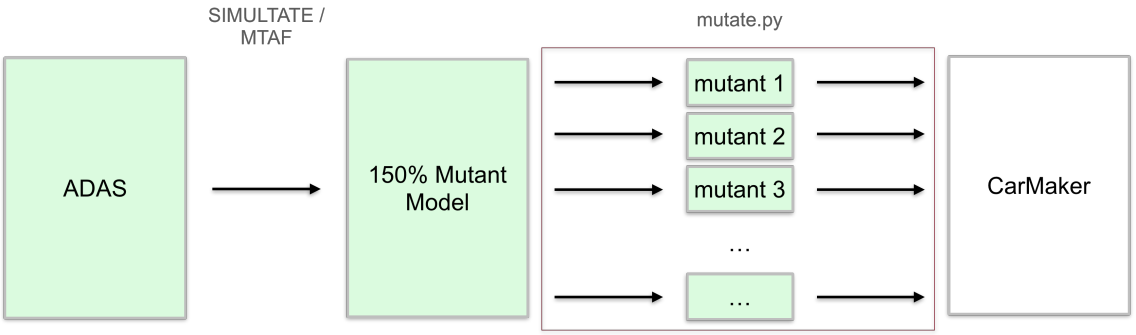
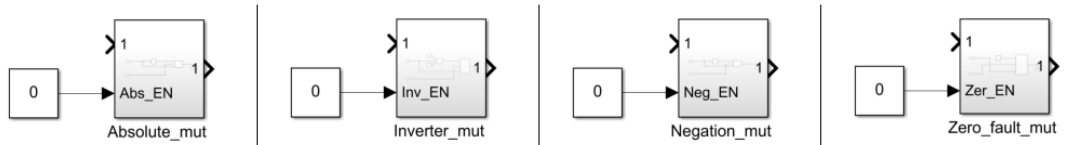


Figure 4.7: Mutant generation workflow

4.2.1 Generation of Mutant Models

In Section 3.2 we mention, that we focus on ADAS that are available within a model-based format. Mevenkamp [Mev19] works on the tool MTAF, which stands for Mutation Testing And Fuzzing. Mevenkamp designs MTAF to test autonomous vehicles and focuses on a model-based mutation for Simulink. Mevenkamp uses the tool SIMULTATE [PRWN16] to mutate Simulink models resulting in a 150% mutant model and the related list *Available_mutations.m* that contains all inserted mutation operators. SIMULTATE provides various mutation operators (see Figure 4.8) such as Absolute_mut (a), Inverter_mut (b), Negation_mut (c), and Zero_fault_mut (d) [PRWN16]. Beyond that, we can define where we want to insert the mutation operator. SIMULTATE provides possibilities such as *inserting_before*, *replacing*, and *exchange_signals* [PRWN16].



(a) Absolute Value (b) Inverter (c) Negation (d) Zero Value

Figure 4.8: Mutation operators according to [PRWN16]

https://doi.org/10.24355/dbbs.084-202102191146-0

However, SIMULTATE does not support the mutation of CM4SL models. CM4SL models are specific Simulink models, that connect with the CarMaker simulation environment. Figure 4.9 presents the mutant model of an AEB-system. Each orange element directly connects to CarMaker transferring sensor data to the model and shift calculated values in the form of variables back to CarMaker. Mevenkamp adapts SIMULTATE to mutate CM4SL models thus MTAF supports their mutation. Beyond the mutant generation, MTAF also executes the mutation testing and evaluation. Executing the tests, MTAF controls Simulink due to a python interface, thus Simulink indirectly controls CarMaker to execute the test cases [Mev19]. However, we use the mutant model creation part of MTAF, we do not aim for the execution of mutation testing using MTAF. Creating the 150% mutant model, we implement and compose mutation operators inspired by Mevenkamp [Mev19].

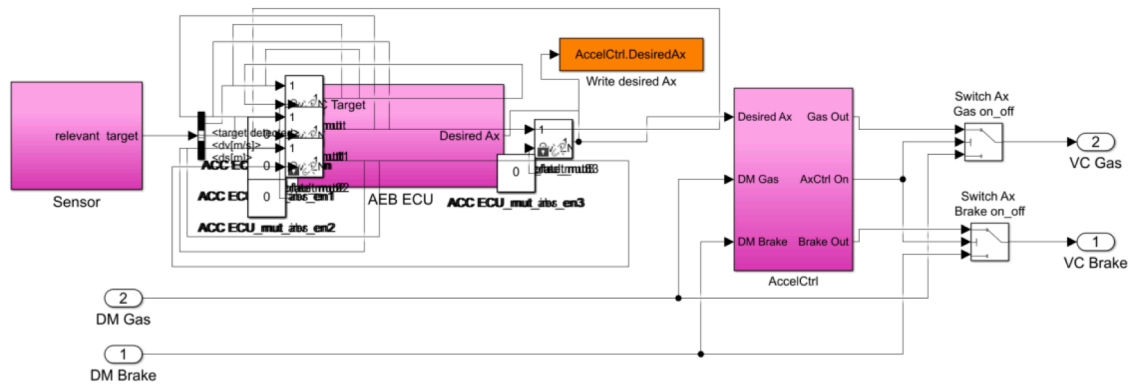


Figure 4.9: Mutant model of an AEB CM4SL model inspired by [Arc18]

4.2.2 Integration of ADAS into Simulation Environment

There are three possibilities to integrate ADAS into the CarMaker simulation environment [IPGb]. One approach is the *integration of C-Code models* [IPGb]. Thereby, we implement the ADAS using CarMaker specific C-Code functions. IPG Automotive GmbH describes these functions within their documentation [IPGb]. IPG Automotive GmbH also provides examples and template data to create the CarMaker specific C-Code models. We need to update the CarMaker makefile, register the resulting source code, and rebuild the CarMaker engine. Using this interface, we can activate up to ten independent ADAS in parallel. We need to register each ADAS we want to activate, but we can register more than ten ADAS.
Beyond that approach, CarMaker provides the interface *CarMaker for Simulink* for a connection to Simulink [IPGb]. When we use this interface, we can implement model-based ADAS. IPG Automotive GmbH provides CarMaker Simulink blocks that we can integrate into our model, thus we create a CM4SL model and connect Simulink and CarMaker. Beyond that, CarMaker provides utilities for Matlab.
The third approach, *Simulink Coder Interface*, combines the two interfaces mentioned above [IPGb]. Therefore, we create CM4SL models such as we do using the *CarMaker for Simulink* interface. Differently, we do not start the simulation using Simulink next to CarMaker, but we export the CM4SL model using a C-Coder. In

https://doi.org/10.24355/dbbs.084-202102191146-0

the next step, we can register the resulting C-Code and rebuild the CarMaker engine. This approach also allows up to ten independent ADAS within a simulation. Within this thesis, we focus on the *Simulink Coder Interface*. This approach provides the advantages of both, CM4SL and C-Code model integration. We expect high compatibility and availability of ADAS models due to model-based software development, which we mentioned in Section 3.2. Beyond that, we can execute our simulations independently of Simulink. CarMaker contributes a better performance using C-Code models, according to IPG Automotive GmbH [IPGb].

We use vehicle-files to integrate the registered C-Code models into the scenario. According to IPG Automotive GmbH [IPGc], a vehicle-file bases on the InfoFile-syntax and contains various parameters to describe the Ego vehicle. For example, there are parameters for suspension, sensors, and VehicleControl within this file. VehicleControl modules represents ADAS. Up to ten prioritized VehicleControl modules are possible. The first VehicleControl module has the highest priority. However, we create a vehicle-file template containing keys that are surrounded by $-signs. Listing 4.2 presents an excerpt of our vehicle-file template, where we use an AEB and a key for another ADAS in parallel.

```
...
## Vehicle Control ###################################
VehicleControl.0.Kind = AEB 1
VehicleControl.1.Kind = $vehicleControlTemplate$ 1
VehicleControl.2.Kind =
VehicleControl.3.Kind =
VehicleControl.4.Kind =
VehicleControl.5.Kind =
VehicleControl.6.Kind =
VehicleControl.7.Kind =
VehicleControl.8.Kind =
VehicleControl.9.Kind =
VehicleControl.Comment:
...
```

Listing 4.2: Extract of a vehicle-file

4.2.3 Generation and Integration of Mutant Test Suites into the Simulation Environment

There are mainly two reasons to automate the mutant generation and integration into the simulation environment. First, an automation enables traceability if the automation bases on rule-based algorithms. Second, we can handle a large number of mutants without a minimum of human resources. However, we split the automation of mutant generation and integration into the simulation environment into two key sections. The first section describes our python script *mutate.py* that generates mutant suites based on the 150% mutant model. The script also exports the resulting mutants into C-Code models. In the second section, we present a bash script *model_registration.sh* to integrate each mutant model into the simulation environment. CarMaker require, that we register each model before we use it within a simulation.

https://doi.org/10.24355/dbbs.084-202102191146-0

Mutant suite generation: mutate.py

In Figure 4.10, we present the workflow of the mutate.py script. The script mainly generates C-Code mutant models, generates vehicle-files, and makes an entry in the *Database_mutant.csv*. The export destination of the C-Code models is the *src* folder within a CarMaker project.

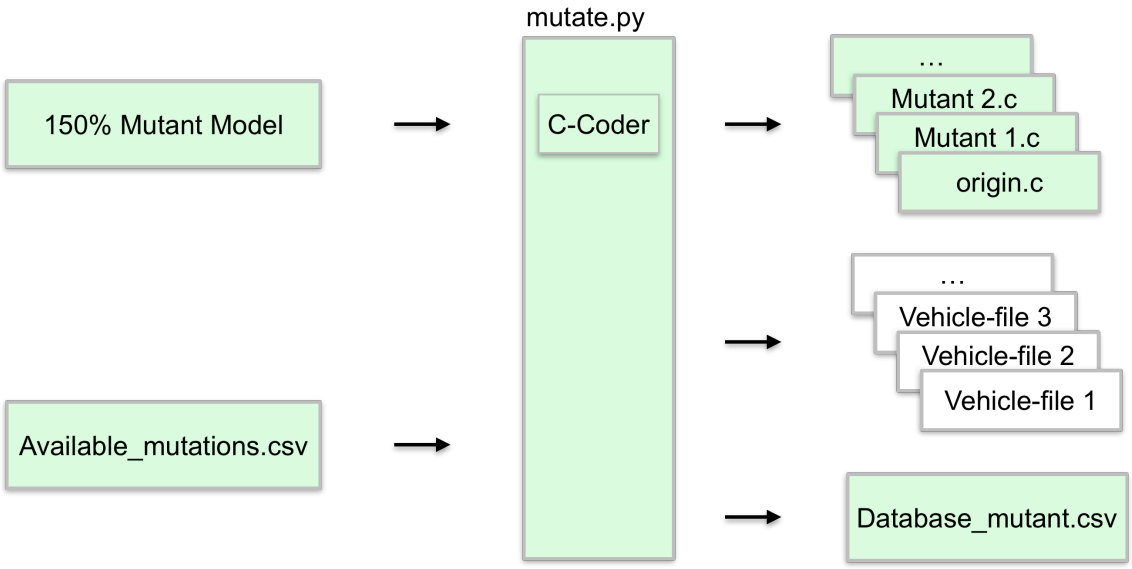


Figure 4.10: Workflow of mutate.py script

We parse the list *Available_mutations.m* generated from SIMULTATE / MTAF into CSV-format. For the mutate.py script, we have the 150% mutant model and the related list *Available_mutations.csv* as input. The mutate.py script opens a Simulink-session to connect to the 150% mutant model. In a first step, we extract the original ADAS model from the 150% mutant model. Therefore, we export the model, without activation of any mutation operator, using the Simulink C-Coder. We also generate a vehicle-file from the vehicle-file template and connect the C-Code model as VehicleControl element by replacing the corresponding keys in the vehicle-file template.

Beyond that, we create the mutant models. For each mutant, we copy the Simulink model and rename it. According to *Available_mutations.csv* list, we activate a mutation operator in the Simulink-model, by changing the enable flag. Thereby, we focus on single order mutants. The mutate.py script also enables the possibility to integrate various mutation strategies we mention in Section 3.2. We can implement the mutation strategies within the mutate.py script according to the *Available_mutations.csv* list. However, the enable flag within the 150% mutant model is designed by a constant. If the constant has the value 0, we disable the mutation operator. Else, if the value is 1, we enable the mutation operator. We export the mutant model using the Simulink C-Coder resulting in a C-Code model. As previously explained for the original model, we also implement the mutant model into the vehicle-files. We generate one vehicle-file for each mutant.

https://doi.org/10.24355/dbbs.084-202102191146-0

The mutate.py script also generates the database *Database_mutant.csv*. This database contains entries that link the orginal ADAS model and all available mutants models to the related vehicle-files. As we explained in Section 4.1.2, we use the *Database_mutant.csv* to integrate vehicle-files into the InfoFiles. Therefore, we have to generate all mutants *before* we execute the transfer from configuration-files to InfoFiles. Beyond that, we have to update all scenario InfoFiles, if we change the mutation. We apply each scenario to each mutant.

Mutant suite integration: model_registration.sh

After the mutation process, we need to register the resulting C-Code models and rebuild the CarMaker engine. If the C-Code model export destination of mutate.py varies from the *src* folder within the CarMaker project, we have copy all C-Code models there. The Simulink Coder creates a folder for each C-Code model export. Within the model export folder, there is the source code as well as a *.mk*-file to build the model. However, for the model registration, we build the model using a GCC compiler. The compiler is designed for UNIX environments, IPG Automotive GmbH suggests to use *MinGW/MSYS* [MSY] for Windows platforms. IPG Automotive GmbH provides a modified version of MSYS [IPGb]. For this purpose, we implement a bash script(see Listing 4.3). Our bash script executes the *.mk*-file of each mutant model. Executing the *.mk*-file, we also set up the model in the CarMaker makefile. After model registration, we need to rebuild the CarMaker engine. Therefore, IPG Automotive GmbH [IPGb] instructs to run *make* in the *src* folder within the CarMaker project directory using MSYS.

```
1  #!/bin/bash
2
3  echo Start make of each VehicleControl mutant
4
5  # change to directory where shell script is located
6  cd `dirname $0`
7
8  # look into each folder
9  for i in *rtw; do
10     cd $i
11     # extract model name
12     echo Excecute make to ${i%?????????????}.mk
13     make -f ${i%?????????????}.mk
14     cd ..
15 done
```

Listing 4.3: Bash script for mutant registration

4.3 Safety Envelope Controller

According to our concept we presented in Section 3.3 we use a SEC to evaluate the behavior of an ADAS. We aim for a model-based SEC for the same reasons as we use a model-based ADAS. Mevenkamp [Mev19] also uses a model-based SEC. He implements the SEC into the original ADAS model before creating the 150% mutant model. During the mutation process, he suppresses each mutation operator, that is

https://doi.org/10.24355/dbbs.084-202102191146-0

located within the SEC. We aim for a strict separation of ADAS and SEC. In contrast to Mevenkamp, we implement the SEC in a separate Simulink model. According to our approach of using the Simulink Coder Interface of CarMaker, we can activate up to ten independent ADAS in parallel (see Section 4.2.2). For this purpose, we need to register each model and activate the ADAS using the VehicleControl section within the vehicle-files. We export the SEC from Simulink and includes it into each vehicle-file as the first VehicleControl module. Thus, the SEC is the vehicle control element with the highest priority [IPGc]. Concurrently, the SUT forms the second VehicleControl module. Refering to Section 4.2.3, the python script *mutate.py* and the bash script *model_registration.sh* also handle the generation and registration of the SEC. The generation and registration of the SEC in CarMakeris identical with the generation and registration of mutants except the entry into the *Database_mutant.csv*. There is no need to add the SEC to the *Database_mutant.csv*.

The implementation of our SEC model is presented in Figure 4.11. The output *SafetyOverall.OK* gives the vehicle's safety state according to defined criteria of input data. We add a collision detection sensor which is provided by the CarMaker simulation tool. The related safety criterion implies that we do not detect any collision. Further inputs are undefined within our implementation, but might be extended with little effort.

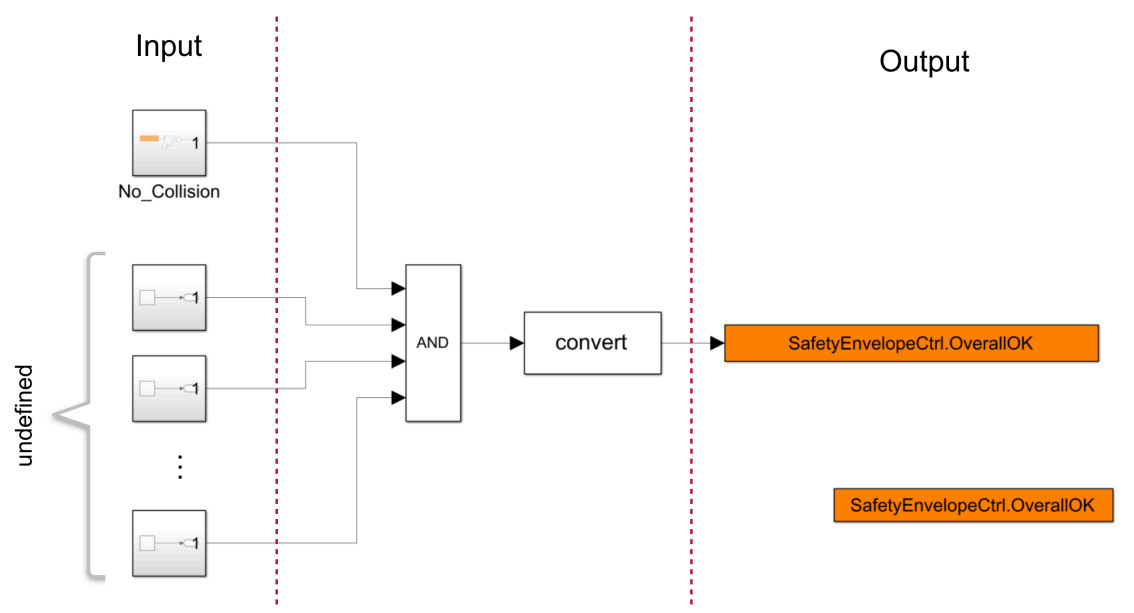


Figure 4.11: Implementation of safety envelope controller

4.4 Simulation

There are several tools to perform simulations within the automotive context. For example, there are CarMaker by IPG Automotive GmbH [IPGa], or Virtual Test Drive (VTD) by VIRES Simulationstechnologie GmbH [VIRb] and an open-source project CARLA [DRC+17]. While both, CarMaker and VTD are commercial simulation solutions, CARLA is an open-source simulation tool. In Section 3.4, we

https://doi.org/10.24355/dbbs.084-202102191146-0

define requirements that we have to the simulation tool. One requirement mention a comprehensive documentation. CARLA provides freely available documentation. The full documentation for the tools CarMaker and VTD is not freely available, but we have access to a CarMaker license due to preliminary research. In the following, we focus on CARLA and CarMaker. Another requirement that we have to the simulation tool is an interface to implement ADAS models that are available in a model-based format. CarMaker provides an interface to implement ADAS model that are available as C-Code, an interface to connect CarMaker to Simulink, and an interface to implement C-Code models that are generated using a Simulink C-Coder [IPGb]. For CARLA, a Matlab-CARLA Interface is available[1] and provides a connection to Simulink via ROS interface or python interface. Beyond that, we aim for a simulation tool that provides test case management for efficient testing. CarMaker provides the CMIT TestManager to execute TestRuns automatically as well as evaluate the TestRuns according to certain criteria [IPGd]. For CARLA, we can control the VVE simulation using python scripts[2] thus we can automate testing due to self-developed scripts. Thus both, CarMaker and CARLA fulfill the requirements we have for the simulation tool. However, CarMaker provides both, a GUI and a ScriptControl interface to control the simulation thus, the handling of CarMaker is more user-friendly than the handling of CARLA.

CarMaker contains a TestManager component for test automation providing a GUI [IPGd]. The TestManager combines input and output data according to Figure 4.12. An InfoFile represents the scenarios that we want to simulate. The mutant elements represents the SUT and the SEC describes the criteria to evaluate the simulation. The output of the CarMaker TestManager indicates whether the vehicle operates according to its specifications or not.

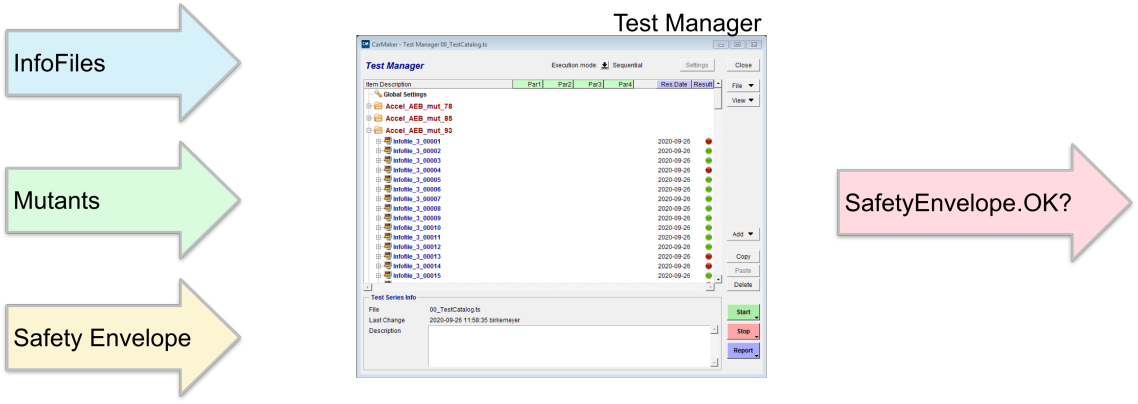


Figure 4.12: Input and output data structure for simulation tool

The TestManager requires a set of TestRuns. A TestRun represents a scenario using CarMaker InfoFile syntax. The TestManager executes the TestRuns automatically and provides the potential to save time. For example, we can execute the TestRuns using high-performance computing to test several TestRuns in parallel [IPGd]. For

[1] https://github.com/darkscyla/MATLAB-Carla-Interface#getting-started
[2] https://carla.readthedocs.io/en/latest/

https://doi.org/10.24355/dbbs.084-202102191146-0

test execution using the CarMaker TestManager, we need a TestSeries-file. We generate the file *TestCatalog.ts* using the transformation tool that we present in Section 4.1.2. The TestSeries-file contains the configuration of the automated test case execution. Listing 4.4 gives an excerpt of *TestCatalog.ts*. Within the TestSeries-file, we implement the criteria to evaluate a TestRun. Beyond the definition of the criteria, the TestSeries-file determines the order of TestRun execution. We structure the order of TestRun execution using groups. Each group contains all TestRuns linked with the same mutant. In Listing 4.4 we present the excerpt of a group definition and the configuration of two TestRuns. We define a Group in Line 3 and name the group according to the mutant model in Line 4. The configuration of the first Test Run within this group starts in Line 6 and 7. We specify the criteria to evaluate the test case from Line 8 to 12.
However, we need to define a criterion that evaluates each TestRun. We make use of our SEC. We link the SEC with the highest-priority VehicleControl module using a C-Code model exported from Simulink. The output of the SEC results in a CarMaker/Simulink block. This block transfers information that we calculate within the model into the CarMaker simulation environment. We use the binary output data of the SEC within the TestManger to define whether the vehicle operates according to its specifications or not. If the vehicle operates according to its specification, the safety overall ok output is 1, otherwise the output changes to 0. However, the CarMaker Test Manager can classify TestRuns into *good*, *warn*, and *bad*. We focus on the classification in *good* and *bad*. *Good* means, the vehicle operates according to the specification during the whole TestRun, otherwise we classify it as *bad*.

```
1        ...
2        Step.0 = Settings
3        Step.0.Name = Global Settings
4        Step.1 = Group
5        Step.1.Name = AccelCtrl_mutationBlocks_mut_74
6        Step.1.0 = TestRun
7        Step.1.0.Name = Infofile_1_00001
8        Step.1.0.Crit.0.Name = Criterion 0
9        Step.1.0.Crit.0.Description:
10       Step.1.0.Crit.0.Good =
11       Step.1.0.Crit.0.Warn =
12       Step.1.0.Crit.0.Bad = [get SafetyEnvelopeCtrl.OverallOK] == 0
13       Step.1.1 = TestRun
14       Step.1.1.Name = Infofile_1_00002
15       Step.1.1.Crit.0.Name = Criterion 0
16       Step.1.1.Crit.0.Description:
17       Step.1.1.Crit.0.Good =
18       Step.1.1.Crit.0.Warn =
19       Step.1.1.Crit.0.Bad = [get SafetyEnvelopeCtrl.OverallOK] == 0
20       ...
```

Listing 4.4: Excerpt of a TestSeries-file for CarMaker TestManager (input)

4.5 Evaluation using Mutation Score

We present the structure of our evaluation workflow in Figure 4.13. From left to right, we export the simulation results from the TestManager and store them into

https://doi.org/10.24355/dbbs.084-202102191146-0

a database. Subsequently, we select various datasets of the database, calculate the mutation score and evaluate the data.

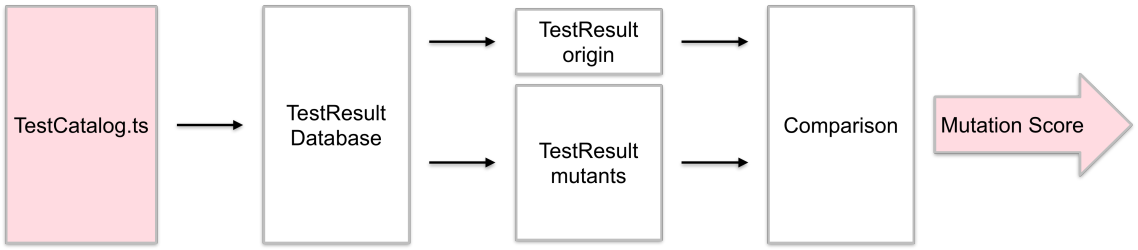


Figure 4.13: Evaluation workflow

The TestManager provides two possibilities to extract the simulation results [IPGd]. We can save the simulation results within the TestSeries-file. Otherwise, we can export a PDF-document containing a detailed test report. Saving the simulation results into the TestSeries-file, the TestManager adds some lines into the current TestSeries-file for each TestRun. According to our TestSeries-file example in Listing 4.4, we present the same extract of the TestSeries-file after TestRun execution in Listing 4.5. In the TestSeries-file after the test execution, there are four new lines for each TestRun. For instance, the overall result for the first TestRun is given in line 13. Within this TestRun the result is good thus the vehicle operates according to the specification.

However, we implement a tool to extract the overall simulation result for each TestRun and transfers the simulation result to a database collecting various simulation results. We realize this tool in Java and create a GUI to guide the user selecting the current TestSeries-file. This tool extracts the simulation results from the TestSeries-file and inserts the simulation result data into the database *TestResult.csv*. In Table 4.4, we present an excerpt. We store, for instance, the configuration, the sampling algorithm, mutant model, and a FM identification (fm_ID) within the database. Thus our results in the database are traceable.

Based on this TestResult database, we evaluate the results. We use the programming language R in combination with the IDE R-Studio OpenSource Edition for evaluation. R is a programming language optimized for statistical calculations and commonly used for data science activities [R C20]. We separate the simulation results of original ADAS model and those, which results from simulations with mutant models. We store the simulation results of both, within separate data frames. The simulation results of the original ADAS defines the expected output for the test case. In the next step, we compare simulation results of the mutants regarding the expected output. We join the same test input data according to the configuration and fm_ID. Based on the comparison of the output from mutant and the expected output, we determine whether a test case passes or fails. We calculate the mutation score according to Equation 3.1. We define a mutant as killed if at least one test case fails thus one simulation result of a mutant model varies according to the expected output. Moreover, the TestResult database provides various metadata for each simulation. Thus we can use the TestResult database as a configurable system

https://doi.org/10.24355/dbbs.084-202102191146-0

```
...
Step.0 = Settings
Step.0.Name = Global Settings
Step.1 = Group
Step.1.Name = AccelCtrl_mutationBlocks_mut_74
Step.1.0 = TestRun
Step.1.0.Name = Infofile_1_00001
Step.1.0.Crit.0.Name = Criterion 0
Step.1.0.Crit.0.Description:
Step.1.0.Crit.0.Good =
Step.1.0.Crit.0.Warn =
Step.1.0.Crit.0.Bad = [get SafetyEnvelopeCtrl.OverallOK] == 0
Step.1.0.Result = good
Step.1.0.ResDate = 1600274354
Step.1.0.ManLst = 0 1
Step.1.0.Crit.0.Result = good
Step.1.1 = TestRun
Step.1.1.Name = Infofile_1_00002
Step.1.1.Crit.0.Name = Criterion 0
Step.1.1.Crit.0.Description:
Step.1.1.Crit.0.Good =
Step.1.1.Crit.0.Warn =
Step.1.1.Crit.0.Bad = [get SafetyEnvelopeCtrl.OverallOK] == 0
Step.1.1.Result = good
Step.1.1.ResDate = 1600274366
Step.1.1.ManLst = 0 1
Step.1.1.Crit.0.Result = good
...
```

Listing 4.5: Excerpt of a TestSeries-file for CarMaker TestManager (output)

to evaluate the simulations. For instance, we can investigate which FM or feature lead to a higher mutation score. Beyond that, we can investigate the impact of various mutants on the mutation score.

https://doi.org/10.24355/dbbs.084-202102191146-0

config	**sampling**	**fmID**	**mutant**	**result**	
Simple_right2left;...	ICPL T2	1	Accel_AEB_mut_78	bad	
Innocent_slow_right2left;...	ICPL T2	1	Accel_AEB_mut_78	good	
Approaching_crossing_obj;...	ICPL T2	1	Accel_AEB_mut_78	good	
Approaching_crossing_obj;...	ICPL T2	1	Accel_AEB_mut_78	bad	
Simple_right2left;...	ICPL T2	1	Accel_AEB_mut_78	good	

Table 4.4: Excerpt of *TestResult.csv*

https://doi.org/10.24355/dbbs.084-202102191146-0

5. Evaluation

We aim to investigate sampling strategies for generating scenarios for simulation based validation of ADAS. For this purpose, we set up a process chain to create scenario suites due to sampling on a FM. We use a FM that we derive from a given simulation tool for our evaluation. Subsequently, we execute the scenarios of a scenario suite within a simulation tool. According to assess a scenario suite, we aim to use mutation testing. We create mutants of an ADAS and use the mutants as well as the original function as SUT. For rating the scenario suite, we calculate a mutation score. The higher the mutation score, the stronger the scenario suite. Within this chapter, we provide an insight into our experiments, the subject system, and the results of our experiments. Beyond that, we discuss our findings and their validity.

5.1 Research Questions

Our thesis focuses on the approach of sample-based scenario suite generation. We use mutation testing to evaluate the resulting scenario suites. In Figure 5.1, we present the influences we identify within our process chain on the mutation score. These influences include the scenario creation based on the combination of *FM* and *sampling algorithm* as well as the *mutant test suite* and the *SEC*.

Within our evaluation, we investigate, whether FMs are useful to generate scenario suites automatically. For this purpose, we examine the influence of the FM and the sampling algorithm on the mutation score. Considering the composition of the generated scenario suites, we aim for knowledge regarding the sampling strategy. This view of the composition focuses on single scenarios. In the next step, we investigate the scenarios in more detail, due to rating single features according to the mutation score. We aim for knowledge regarding the FM design. We define the following research questions to evaluate whether FMs are useful to generate scenario suites automatically:

https://doi.org/10.24355/dbbs.084-202102191146-0

Figure 5.1: Influences on mutation score

RQ 1: **Are feature models and sampling useful to generate scenario suites automatically?**
RQ 1.1: What is the impact of the feature model and the sampling strategy?
RQ 1.2: How does the composition of a scenario suite look like?
RQ 1.3: How relevant are single features on the mutation score?

Beyond that, we investigate whether the mutation score is a suitable metric to rate scenario suites. For this purpose, we examine the detection rate of single mutants using various scenarios suites. This evaluation results in an insight into the correlations of mutant detection and sampling strategy. We define the following research question to evaluate whether the mutation score is a suitable metric to rate scenario suites:

RQ 2: **Is the mutation score a practical metric to rate scenario suites?**
RQ 2.1: What is the correlation between mutant test suite and sampling strategy?

5.2 Experiment Design

For the evaluation of the research questions, we use our toolchain to generate scenarios automatically. Subsequently, we execute the simulation of the generated scenarios in combination with the original ADAS or mutant models as SUT and the SEC to evaluate each simulation. We use CarMaker of IPG Automotive GmbH as the simulation tool thus we implement the scenarios in the form of CarMaker specific InfoFiles. The scenarios, ADAS / Mutants, and SEC build the CarMaker input data. The CarMaker TestManager realizes automated testing. The SEC gives an indication, whether the overall system including the ADAS / Mutant performs according to its specification. We use this indication of the SEC as the output of each simulation.

As a first step, we execute simulations with varying input data. For instance, we can use various strategies for scenario or mutant generation as well as their combi-

https://doi.org/10.24355/dbbs.084-202102191146-0

nation. We enter each simulation result, including metadata, into a database (see Section 4.5). In a second step, we evaluate the simulation results. We can select and merge results by using methods from data science. As a metric, we use the mutation score within our experiments. Therefore, we compare the simulation results of mutant ADAS with the expected ouput. The expected output is defined by the simulation results of the original ADAS providing the same metadata. We focus our evaluation on a comparison of different scenario suites. Thus we investigate the influence of various sampling algorithms that we use to create the scenario suites automatically.

5.3 Subject System

In this section, we describe the composition and origin of the input data for the simulation. As input data, we identify according to Figure 4.12 the components: *Scenario*, *ADAS/Mutants* and the *SEC*. We structure this section according to these components.

Scenario

We need a FM as input data to create scenarios. The FM has to implement features of a simulation environment. As described in Section 3.1, we focus on the maneuver integration concept to set up the FM so that we can integrate prepared scenarios into the FM. We use the scenario *AEB_CrossingPedestrianCity* by IPG Automotive GmbH. We implement this scenario as a maneuver template into our FM and rename it to *Simple_Crossing* (see Figure 5.2). According to Section 4.1.1, we realize the FM in FeatureIDE. We identify time-invariant elements. We remove these elements from the scenario and implement them as features. Subsequently, we use the documentation of CarMaker to identify alternatives for each time-invariant element. We combine these elements as features into a common branch of the FM. Figure 5.2 presents the resulting FM (*FM 1*). In parallel, we add the associated CarMaker InfoFile code into our databases. We get the associated Code from the prepared example scenarios or derive it from the CarMaker documentation [IPGb, IPGc, IPGd].

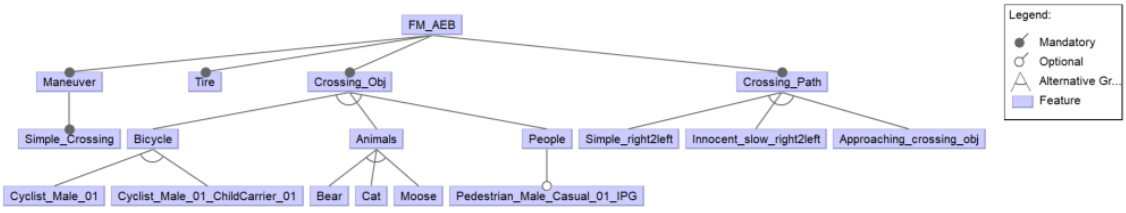


Figure 5.2: FM 1 to generate scenarios for subject system

We create two more FMs (*FM 2* and *FM 3*). For this purpose, we expand the FM 1 presented in Figure 5.2 due to further scenarios and time-invariant elements. Thus FM 1 is part of FM 2 and FM 3 as well as FM 2 is part of FM 3. For instance, we add the EuroNCAP Test Procedures CVNA, CVNC, and CVFA [Eur15a]. Beyond

https://doi.org/10.24355/dbbs.084-202102191146-0

that, we add a scenario, where an object crosses the street at a bus stop and a scenario where a stationary object is on the street behind a curve. In conclusion, we select scenarios that trigger the Autonomous Emergency Braking for Vulnerable Road Users (AEB VRU). AEB VRU protects vulnerable road users such as pedestrians and cyclists due to an emergency braking in a critical situation [Eur15b]. The associated CarMaker scenarios that we use are provided by IPG Automotive GmbH. We integrate the scenarios into the FM. We also add one maneuver *AEB_CrossingCarIntersection* by IPG Automotive GmbH that does not trigger a collision with a vulnerable road user, but with another car. Beyond that, we add environmental elements like wind at various speeds, the position of the sun, and fog. Finally, we create three FMs with different complexity according to the number of features. We present the detailed structure of FM 1 in Figure 5.2, in addition FM 2 and FM 3 in the appendix (see Figure A.1, and Figure A.2).
In FM 3 we implement a trailer as a time-invariant feature. Within some simulations, the trailer sways from side to side. If the trailer sways too strong the simulation runs into an error thus the simulation aborts. Within our evaluation, we exclude these test results. This affects 744 of 76880 simulations we run for FM 3. Thus we exclude less then 1% of the test results due to simulation errors.

We aim to evaluate the influence of various sampling strategies on the simulation result. Therefore, we create multiple scenario suites using various sampling algorithms. For this purpose, we use the product generator of FeatureIDE to extract configurations. We focus on the t-wise sampling algorithms ICPL [JHF12], Chvatal [JHF11], and IncLing [AHKT+16]. The value t is selectable for ICPL $1 \leq t \leq 3$ and for Chvatal $1 \leq t \leq 4$. The value t is $t = 2$ for IncLing according to the definition of IncLing [AHKT+16]. For our evaluation, we choose seven different sampling strategies to generate scenario suites. We focus on ICPL ($t = 1$), ICPL ($t = 2$), ICPL ($t = 3$), Chvatal ($t = 1$), Chvatal ($t = 2$), Chvatal ($t = 3$), and IncLing. Thus results in seven scenario suites that we describe using $ScSu_{Algo}$ where $Algo$ represents the sampling strategy.

$$ScSu_{Algo} = \{S \in ScAll \mid S \, is \, generated \, by \, Algo\}$$

In the following, we name the scenario suites, generated using the sampling algorithms, according to the related sampling algorithm. In Table 5.1 we present the number of scenarios that we generate with each sampling algorithm $|ScSu_{Algo}|$, separated by the three FMs. We execute each scenario in combination with the original ADAS model and each mutant.

ADAS and Mutants

We use an AEB as SUT within our subject system. Such as previously mentioned, we design our FM to generate scenarios, that essentially trigger the AEB VRU. Inspired by Arcidiacono [Arc18] we implement an AEB model into Simulink. Figure 5.3 provides insights into the AEB-system. The AccelCtrl component is taken from CarMaker ACC-model example by IPG Automotive GmbH. In contrast to Arcidiacono, we do not implement the combination of ACC and AEB, thus we slice the given model. Figure 5.4 provides insights into the resulting *AEB-ECU* component. The *AEB-ECU* component, is designed to realize staged braking using two stages.

https://doi.org/10.24355/dbbs.084-202102191146-0

Sampling algorithm	FM 1	FM 2	FM 3
ICPL ($t = 1$)	6	12	10
ICPL ($t = 2$)	19	37	116
ICPL ($t = 3$)	19	37	114
CHVATAL ($t = 1$)	6	12	10
CHVATAL ($t = 2$)	19	37	123
CHVATAL ($t = 3$)	19	37	118
INCLING	19	37	129
Total	**107**	**209**	**620**

Table 5.1: Number of generated scenarios per sampling algorithm

The first stage accelerates the car with $a = -2.45m/s^2$ the second stage accelerates the car with $a = -9.5m/s^2$. Within this thesis, we focus on the second stage. We detach the first braking stage by terminating the output. Beyond that, we add components to ensure a braking to a standstill within the second stage. We realize the input of the AEB system, by an object sensor provided by IPG Automotive GmbH [IPGc, IPGb]. Thus, we focus on the AEB control algorithms, not on the sensor data flow.

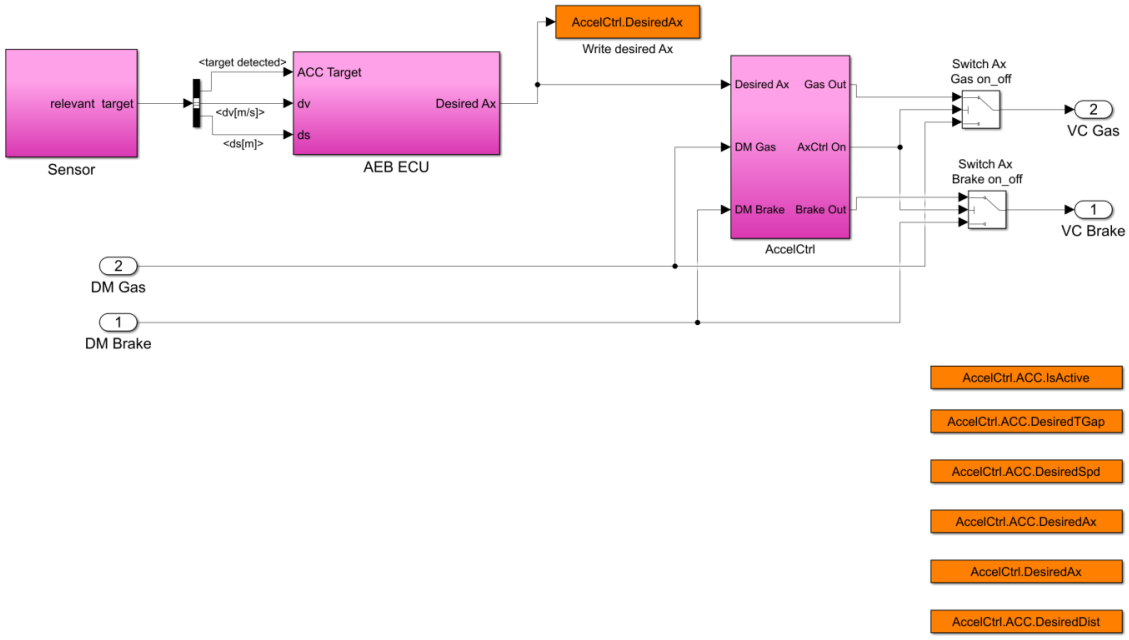


Figure 5.3: Implementation of AEB-ECU model inspired by [Arc18, IPGa]

We create the mutants of the ADAS model within in two steps. In the first step, we generate a 150% mutant model. Subsequently, we extract individual mutants from the 150% mutant model. For the creation of the 150% mutant model, we use SIMULTATE [PRWN16] and MTAF [Mev19] according to Section 4.2. Thereby, we specify the mutation operators and their position. In Table 5.2 we present the

https://doi.org/10.24355/dbbs.084-202102191146-0

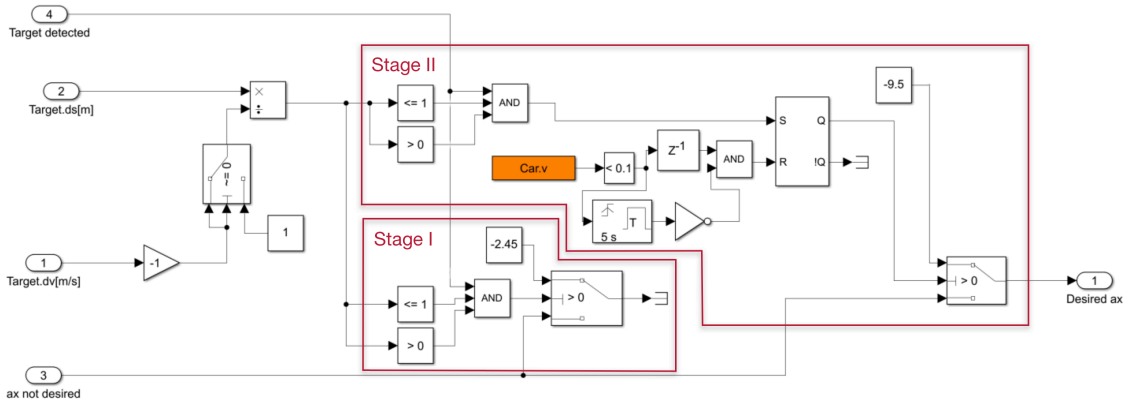


Figure 5.4: Implementation of AEB-ECU model inspired by [Arc18, IPGa]

mutation operators that we use within our experiments, inspired by Mevenkamp [Mev19]. Based on the 150% mutant model we extract individual mutants. For a generation of x mutants, we copy the 150% mutant model x-times and activate various mutation operators within this model.
Within the subject system, we implement 123 mutation operators into the AEB model resulting in a 150% AEB mutant model. From this, we extract first order mutants thus we activate one mutation operator in a mutant model. We get 123 mutant models.

Safety Envelope Controller

According to Section 4.3, we implement the SEC as a Simulink model and connect it to CarMaker using the Simulink Coder Interface. The SEC represents the specifications of the system. We expand the specification of a single ADAS model by an interpretation of the overall situation. We define, that the vehicle performs according to its specification if there is no collision. For this purpose, we implement a collision detection independent of the ADAS model. Figure 5.5 presents the input model module of the SEC. We use the collision detection sensor provided by IPG Automotive GmbH. The sensor returns the number of collisions that proceeds. The SEC considers that the number of collisions is less than one.

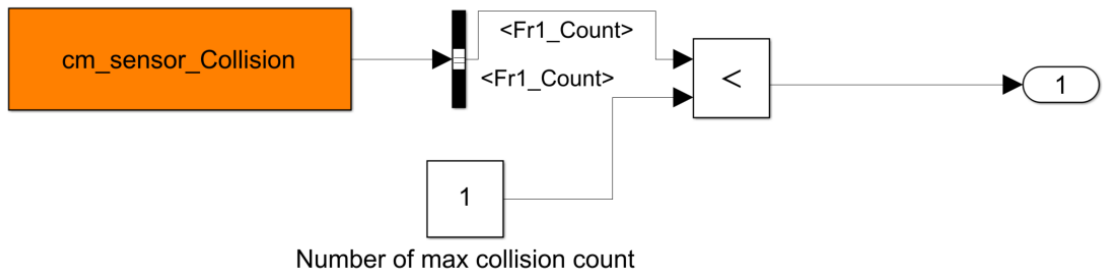


Figure 5.5: Implementation of SEC model

https://doi.org/10.24355/dbbs.084-202102191146-0

Mutation Operator	Referenz Block	Position relative
Absolute_mut	Sum	inserting_before
		inserting_after
	Product	inserting_before
		inserting_after
	SubSystem	inserting_before
		inserting_after
	MinMax	inserting_before
		inserting_after
	Saturate	inserting_before
		inserting_after
	Switch	inserting_before
		inserting_after
Zero_fault_mut	SubSystem	inserting_before
		inserting_after
	Switch	inserting_before
		inserting_after
Inverter_mut	SubSystem	inserting_before
		inserting_after
	Switch	inserting_before
		inserting_after
Negation_mut	MinMax	inserting_before
		inserting_after
	Switch	inserting_before
		inserting_after

Table 5.2: Mutation operators according to [PRWN16, Mev19]

https://doi.org/10.24355/dbbs.084-202102191146-0

5.4 Investigation of the Mutation Score regarding Sampling-Based Scenario Generation

We aim to evaluate scenario suites using the mutation score. We create each scenario suite due to sampling on a FM. Subsequently, we execute a simulation of each scenario in combination with the origin ADAS model and each mutant. We store the simulation results into the TestResult.csv database. Based on the simulation results, we calculate the mutation score.

In the investigation, we calculate the mutation score for the union of an overall scenario suite in Section 5.4.1. We investigate the impact of the FM and the sampling algorithm on the mutation score. In Section 5.4.2, we investigate the influence of single scenarios on the mutation score. Thereby, we calculate the mutation score for each scenario and investigate the composition of a scenario suite according to the sampling algorithms for three FMs. In Section 5.4.3, we investigate the influence of single features on the mutation score. Subsequently, in Section 5.4.4, we investigate the mutant detection rate and the impact of scenario suites on the mutant killing. Thereby, we evaluate which scenario suite kills which mutants.

5.4.1 Influence of Feature Model and Sampling Algorithm on the Mutation Score

We use the three FMs, we presented in Section 5.3 as input for the scenario generation. We expand the FM 1 by adding new maneuvers and time-invariant environmental elements, resulting in FM 2. Similarly, we expand FM 2 resulting in FM 3. Thus FM 1 is part of FM 2 and FM 2 is part of FM 3.
We investigate the complexity of the FMs. We determine the complexity of the FMs according to the number of features. For this purpose, we differentiate between features representing a maneuver and the features representing time-invariant elements of a scenario. In Table 5.3, we present the number of features according to the three FMs. FM 1 is the FM with the lowest complexity, implementing one maneuver feature and nine time-invariant element features. For FM 2 we add three more maneuver features and six more time-invariant element features to FM 1. FM 3 is the FM with the highest complexity implementing three more maneuver features and 35 additional time-invariant element features regarding FM 2.

Feature model	Maneuver features	Time-invariant element features
FM 1	1	9
FM 2	4	15
FM 3	7	50

Table 5.3: Complexity of the feature models

Based on each FM, we generate seven scenario suites using the seven sampling algorithms also presented in Section 5.3. Subsequently, we calculate the mutation score for each scenario suite. We execute the scenarios of each scenario suite in combination with the original ADAS model. The SEC output provides the expected test case result. We apply the same scenario suites on the mutants of the mutant

https://doi.org/10.24355/dbbs.084-202102191146-0

test suite. Subsequently, we define a mutant as killed if the SEC result of the mutant varies from the expected output of the test case within at least one scenario. Thus we evaluate the union of all simulation results within one scenario suite. We use a fixed mutant test suite M. We calculate the mutation score for each sampling algorithm according to Equation 5.1.

$$MS_M(ScSu) = \frac{|\{m \in M \mid m \, killed \, by \, some \, S \, in \, ScSu\}|}{|\{M\}|} \tag{5.1}$$

Figure 5.6 presents the mutation score for each scenario suite generated using the sampling algorithms, separated by the three FMs. The x-axis depicts the sampling algorithms. The y-axis indicates the mutation score. Each scenario suite kills 62 of the 123 mutants for FM 1 and FM 2. Thus, the mutation score is $MS_{FM1\_x} = MS_{FM2\_x} \approx 0,504$ for each combination. In FM 3 the mutation score varies according to the sampling algorithms we use to generate the scenario suites. The scenario suites generated by the feature-wise sampling algorithms Chvatal ($t = 1$) and ICPL ($t = 1$) lead to a mutation score of $MS_{FM3\_Ch\_t1} = MS_{FM3\_ICPL\_t1} \approx 0,472$. Thus the scenario suites generated using feature-wise sampling algorithms of FM 3 result in a mutation score less than on FM 1 and FM 2. We observe the same trend for the scenario suite generated by ICPL ($t = 2$) resulting in a mutation score of $MS_{FM3\_ICPL\_t2} \approx 0,48$. The scenario suites basing on the sampling of Chvatal ($t = 2$), Chvatal ($t = 3$), ICPL ($t = 3$), and IncLing lead to the same mutation score as for FM 1 and FM 2: $MS_{FM3\_Ch\_t2} = MS_{FM3\_Ch\_t3} = MS_{FM3\_ICPL\_t3} = MS_{FM3\_Inc} \approx 0,504$

5.4.2 Influence of Single Scenarios on the Mutation Score

For a better understanding of the influence of sampling algorithms on the scenario suites, we perform a more detailed evaluation than we previously introduced. We calculate the mutation scores as we do in Section 5.4.1 according to Equation 3.1 but, in this evaluation, we calculate the mutation score for each scenario, not for an overall scenario suite. For this purpose, we use Equation 5.2. This evaluation results in a ranking of single scenarios according to the mutation score. Subsequently, we match the scenarios and the resulting mutation scores with the corresponding sampling algorithms.

$$MS_M(S) = \frac{|\{m \in M \mid m \, killed \, by \, S \, in \, ScAll\}|}{|\{M\}|} \tag{5.2}$$

In Figure 5.7 we present a histogram showing the distribution of the mutation score that occur within all scenarios according to the FMs. We present more detailed histograms separated by the FMs and sampling algorithms in the appendix (see Figure A.3). The x-axis depicts the mutation score. The y-axis indicates the number of occurrences. For each FM we identify two peaks within the distribution. One peak is located at a mutation score of $MS_M \approx 0,05$, the second one at a mutation score of $MS_M \approx 0,42$. We classify the data into two classes introducing a threshold of $\theta_{MS} = 0,2$. One class ($MS_M > \theta_{MS}$) contains scenarios leading to a strong

https://doi.org/10.24355/dbbs.084-202102191146-0

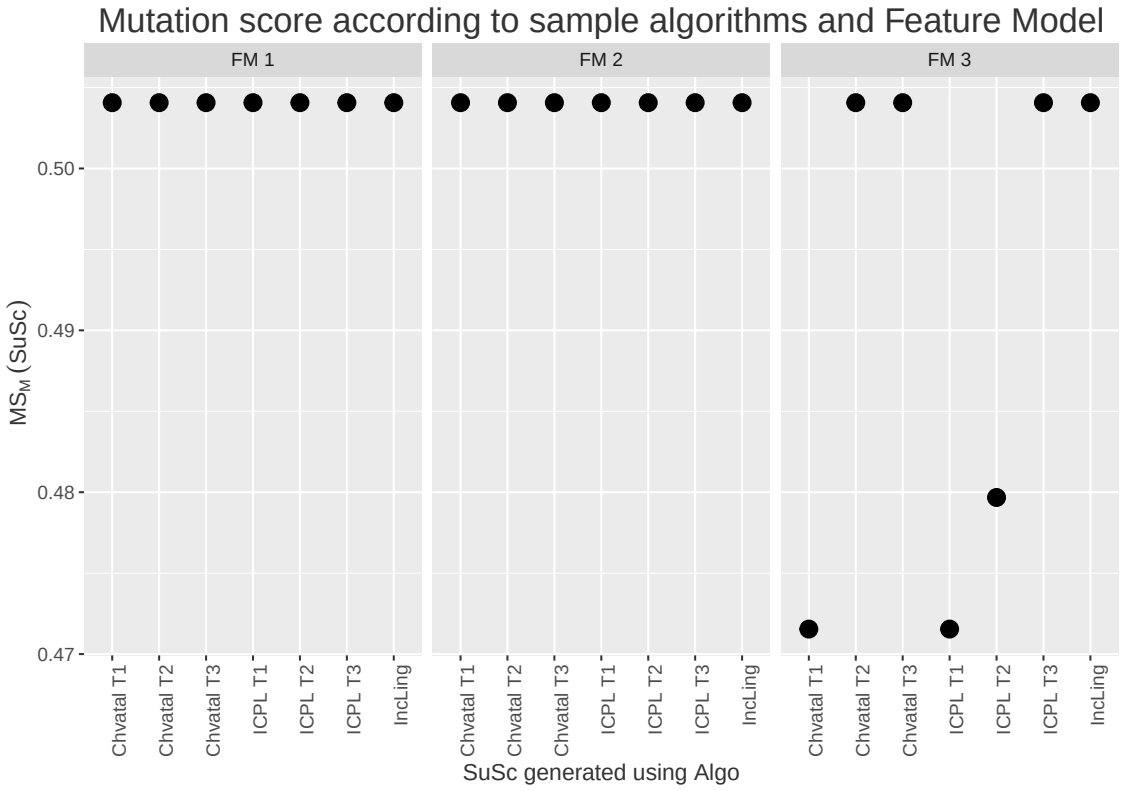


Figure 5.6: Mutation score according to sampling algorithm and FM

mutation score, the other class ($MS_M \leq \theta_{MS}$) contains scenarios that does not lead to a strong mutation score. We mark each class in Figure 5.7.

We investigate the number of scenarios within each class. For this purpose, we consider each FM individually. Table 5.4 presents the ratio of the scenarios in $MS_M \leq \theta_{MS}$ and $MS_M > \theta_{MS}$ for each FM.

Feature model	**Ratio** $MS_M \leq \theta_{MS}$	**Ratio** $MS_M > \theta_{MS}$
FM 1	30,84 %	69,16 %
FM 2	37,8 %	62,2 %
FM 3	78,94 %	21,06 %

Table 5.4: Ratio of scenario distribution per FM

5.4.3 Influence of Single Features on the Mutation Score

We investigate the influence of single features on the mutation score. For this purpose, we choose FM 3 with the highest complexity, as evaluated by the number of features. We calculate the relevance r of each feature using Equation 5.3. For each feature, we count the number of scenarios, where the feature f_i occur and the simulation result of any mutant varies regarding the expected output. We define the corresponding set to $ScSu_{f_i}^{kill(M)}$.

$$ScSu_{f_i}^{kill(M)} = \{S \in ScAll | f_i \in S \wedge S \in kill(m \in M)\}$$

https://doi.org/10.24355/dbbs.084-202102191146-0

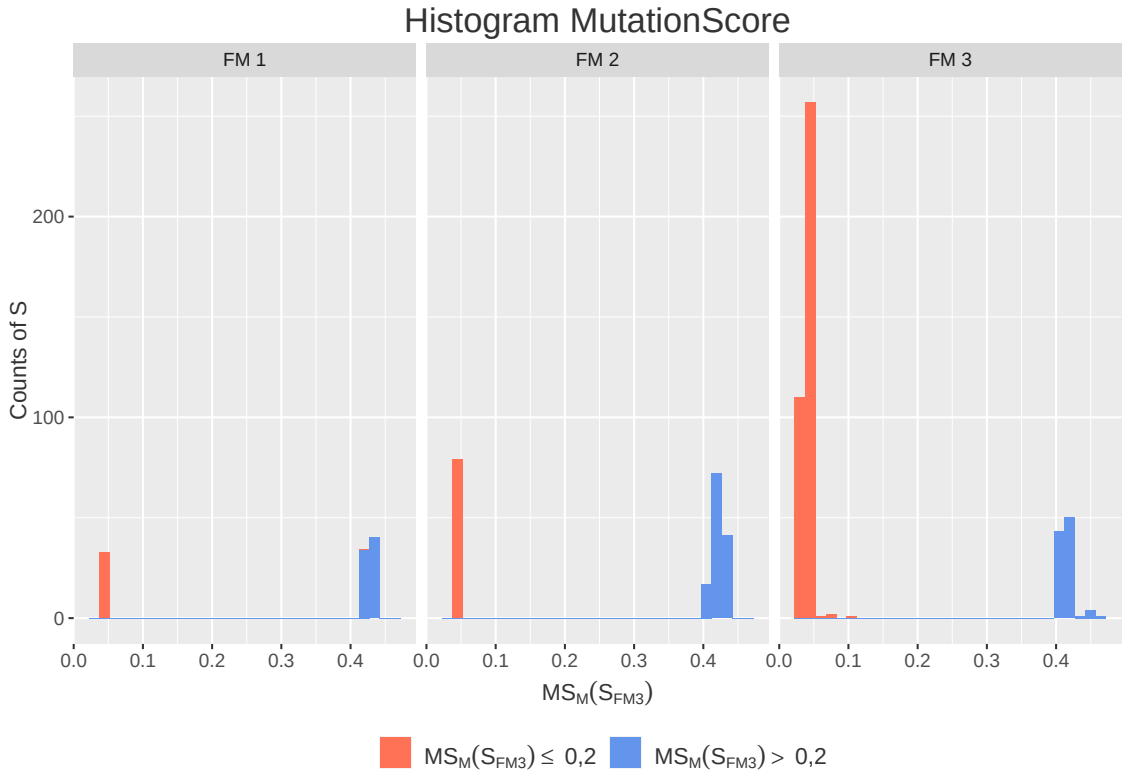


Figure 5.7: Distribution of scenario-based mutation score per FM

We divide the number of these scenarios by the number of all scenarios, where the feature f_i occurs. We define the set of all scenarios, where the feature f_i occurs to $ScSu_{f_i}$.

$$ScSu_{f_i} = \{S \in ScAll |\ f_i \in S\}$$

Within the calculation, we consider only optional, child features and exclude all mandatory or parent features.

$$r(f_i) = \frac{|ScSu_{f_i}^{kill(M)}|}{|ScSu_{f_i}|} \tag{5.3}$$

For our investigation, we separate maneuver and time-invariant features. In Figure 5.8 we present the relevance of the maneuver features. The x-axis depicts the maneuver-features. The y-axis indicates the relevance. We identify the highest relevance of $r \approx 0,15$ of the NCAP maneuver *CVFA*. The three NCAP and the *simple_crossing* maneuver resulting in a relevance $r > 0,12$. The relevance of the maneuver *AEB_StandingMooseRural*, *AEB_CrossingCarIntersection* and *AEB_CrossingPedestrianBusStop* are $r < 0,04$. Within this investigation, *AEB_CrossingPedestrainBusStop* is the maneuver with the minimum relevance.

In Figure 5.9, we present the relevance of time-invariant features of FM 3. The x-axis depicts the maneuver-features. The y-axis indicates the relevance. The color displays the feature categories according to the legend. The feature categories arise

https://doi.org/10.24355/dbbs.084-202102191146-0

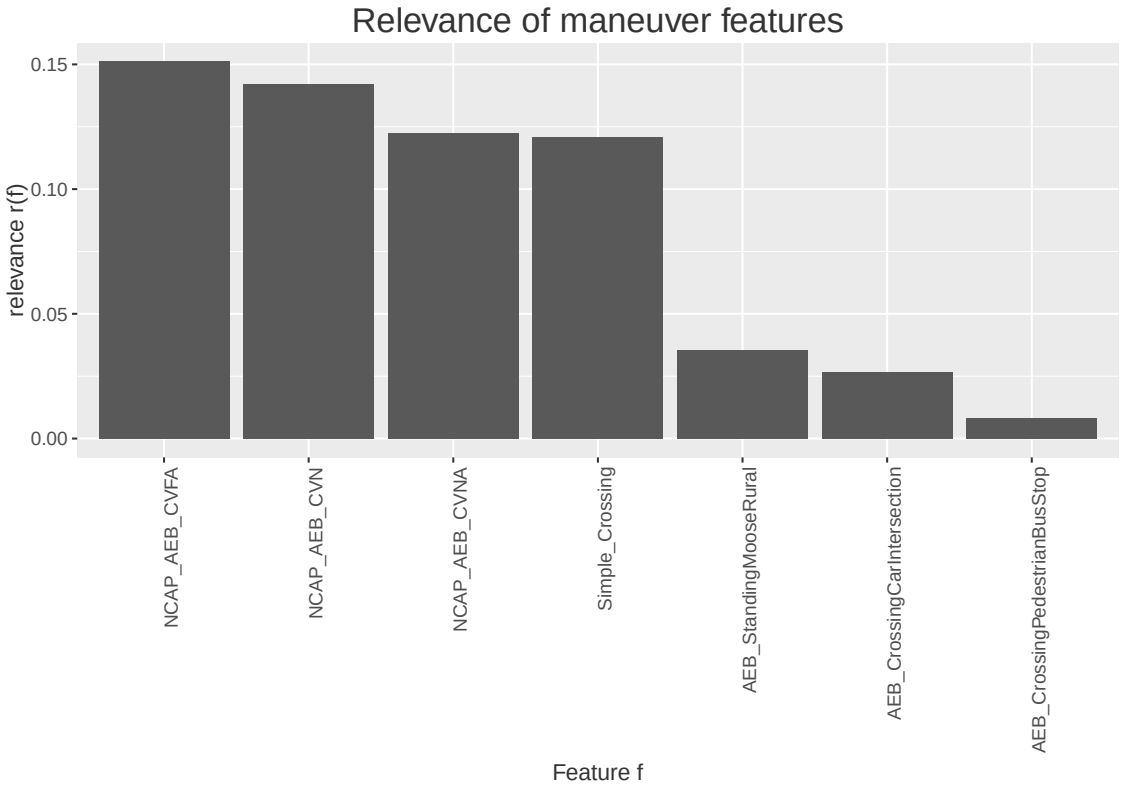


Figure 5.8: Relevance of maneuver features for FM 3

from the parent features of the time-invariant features. We identify, that the relevance of all time-invariant features is located between $r = 0,03$ and $r = 0,18$ ($0,03 \leq r(f_{time-invariant}) \leq 0,18$). Within each feature category, we identify features that lead to a higher relevance than others. Nevertheless, we do not identify single exception-features that lead to a disproportionate relevance. However, within in the *crossing_path*, *Environment Temperature,* and *Speed* feature-categories, we identify a higher variance than in *Environment Sun*, *Environment Fog*, or *Trailer*. Beyond that, we see, that the relevance median of the feature-category *Environment Wind* ($\overline{r}(f_{wind}) \approx 0.15$) is higher than for *Environment Sun* ($\overline{r}(f_{sun}) \approx 0.1$). The median of the *Environment Sun* feature-category, in turn, is higher than for *Crossing_object* ($\overline{r}(f_{crossing\_obj}) \approx 0.08$). The *trailer* features leads to the slightest relevance value on median ($\overline{r}(f_{Trailer}) \approx 0.04$) within this investigation.

5.4.4 Impact of Scenario Suites on Mutant Killing

We investigate the impact of a scenario suite on killing a specific mutant. We calculate the mutant detection rate *mdr* for each mutant in combination with each scenario suite using Equation 5.4. For each mutant, we count the number of scenarios within a scenario suite that kill the mutant. We divide the number of scenarios killing a mutant by the overall number of scenarios within the related scenario suite resulting in the mutant detection rate. The mutant detection rate evaluates how strong a scenario suite is to kill this specific mutant. Beyond that, the mutant

https://doi.org/10.24355/dbbs.084-202102191146-0

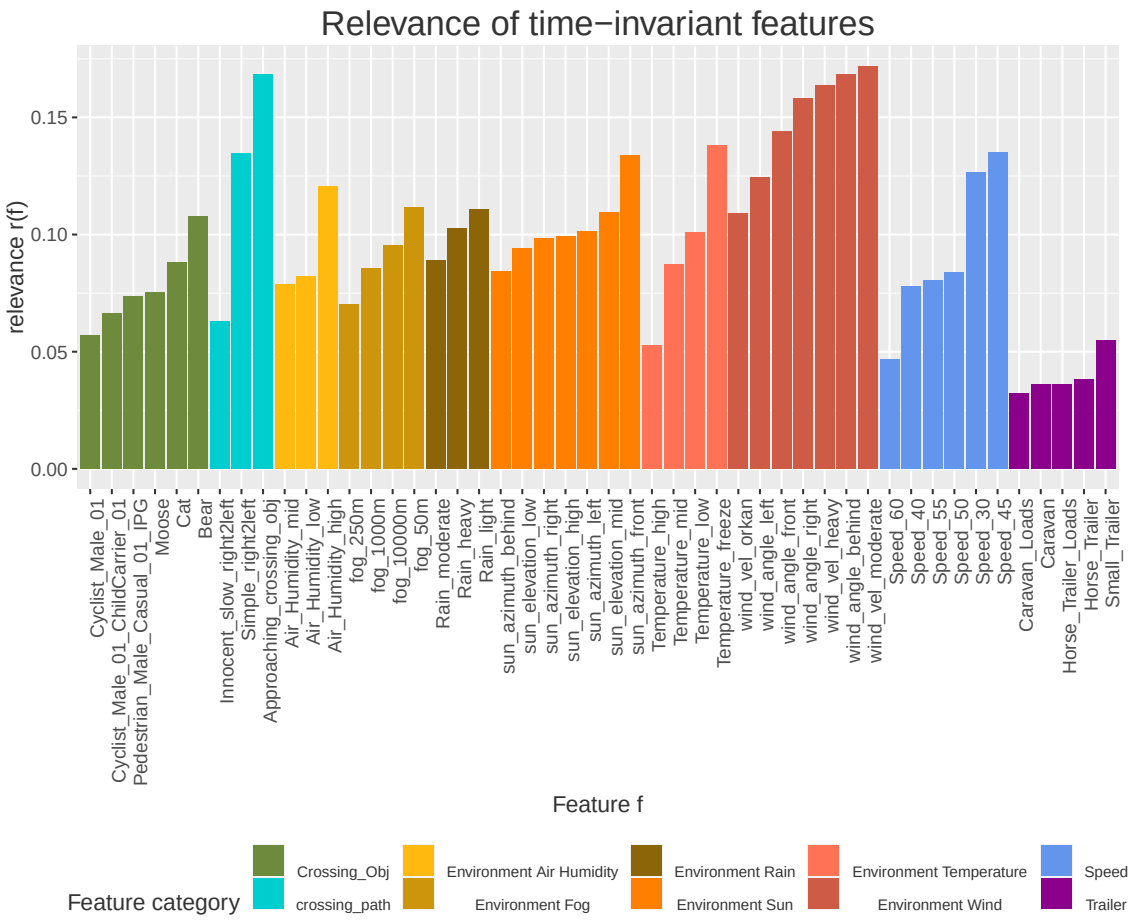


Figure 5.9: Relevance of time-invariant features for FM 3

https://doi.org/10.24355/dbbs.084-202102191146-0

detection rate indicates which scenarios suite detects which mutant. The higher the mdr, the easier to kill the corresponding mutant.

$$mdr(m, Algo) = \frac{|\{S \in ScAll \mid S\ kills\ m \wedge S \in ScSu_{Algo}\}|}{|\{ScAll\}|} \tag{5.4}$$

In Figure 5.10 we present an excerpt of the mutant detection rate mdr for FM 3. We reduce the number of mutants to 14 mutants for better clarity. We present the mutant detection rate for all killed mutants in the appendix (Figure A.4).
We identify easy to kill mutants such as *Accel_AEB_mut_101* or *Accel_AEB_mut_39*. For these mutants, we calculate a mutant detection rate of $0,25 \leq mdr \leq 0,7$, depending on the scenario suite. Beyond that, we identify mutants whose mutant detection rate range is between $mdr = 0,1$ and $mdr = 0,3$ ($0,1 \leq mdr \leq 0,3$) depending on the scenario suite. For instance, this is the case for the mutants *Accel_AEB_mut_102*, *Accel_AEB_mut_103*, and *Accel_AEB_mut_20*. We also identify mutants such as *Accel_AEB_mut_44*, *Accel_AEB_mut_71*, and *Accel_AEB_mut_6* resulting in a very small mutant detection rate of $mdr < 0,02$. Thereby the mutants *Accel_AEB_mut_44*, *Accel_AEB_mut_71* are not detected by any scenario of the scenario suites generated using Chvatal ($t = 1$), ICPL ($t = 1$) and ICPL ($t = 2$). The mutant *Accel_AEB_mut_6* is only detected by the scenario suite generated using ICPL ($t = 2$).

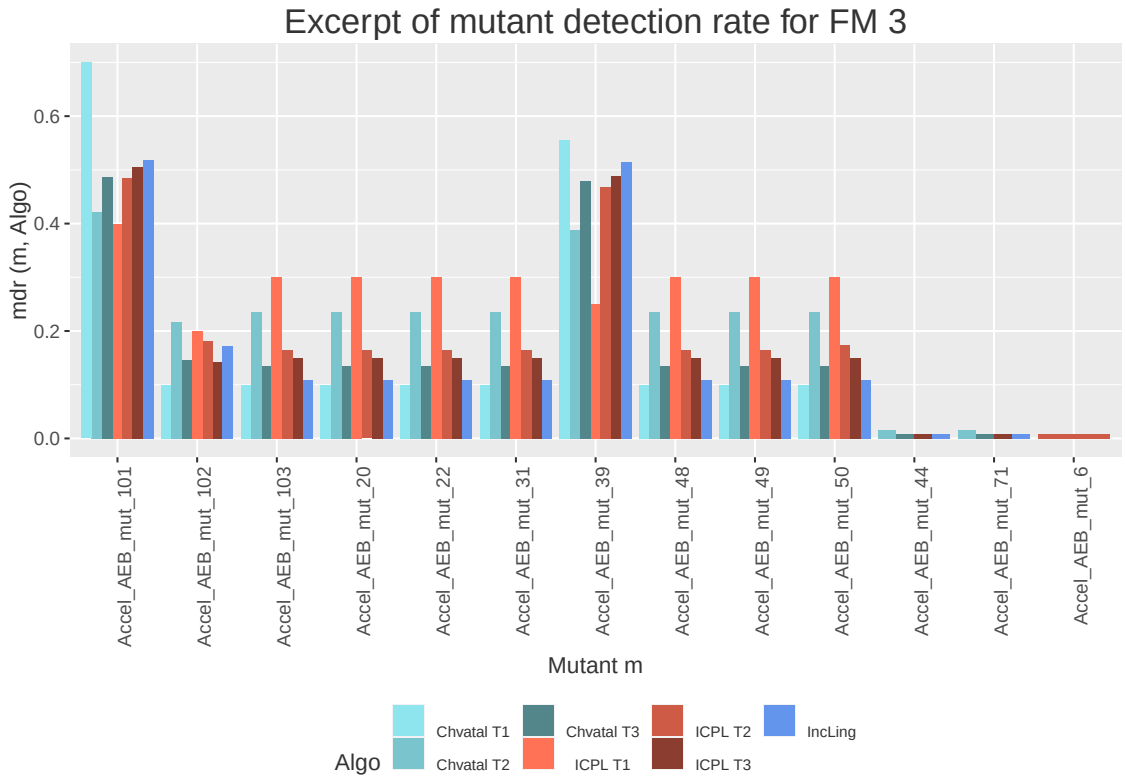


Figure 5.10: Excerpt of mutant detection rate for FM 3

Beyond the investigation of the mutant detection rate regarding single mutants, we can also investigate the performance of the scenario suites considering Figure 5.10

https://doi.org/10.24355/dbbs.084-202102191146-0

and Figure A.4. The mutant detection rate of the scenario suite generated using Chvatal ($t = 1$) results in the highest mutant detection rate for the mutants *Accel_AEB_mut_101* and *Accel_AEB_mut_39*, while ICPL ($t = 1$) leads to the smallest mutant detection rate. Concurrently, ICPL ($t = 1$) results in the highest and Chvatal ($t = 1$) in the smallest mutant detection rate for the mutants *Accel_AEB_mut_103*, *Accel_AEB_mut_20*, and *Accel_AEB_mut_22*. Both scenarios suites generated using ICPL ($t = 1$) and Chvatal ($t = 1$) do not detect the mutants *Accel_AEB_mut_44* or *Accel_AEB_mut_71*. In contrast, the scenario suites generated using ICPL ($t = 3$), Chvatal ($t = 2$), Chvatal ($t = 3$), and, IncLing kill them. Beyond that, we identify, that ICPL ($t = 2$) leads to the only scenario suite that kills *Accel_AEB_mut_6*.

5.5 Discussion

In this section, we discuss the results, we present in Section 5.4 according to our research questions. We structure this section into two parts discussing the research questions RQ 1 and RQ 2. In Section 5.5.1, we discuss whether feature models and sampling are useful to generate scenario suites automatically thus RQ 1. In Section 5.5.2, we discuss whether the mutation score is a practical metric to rate scenario suites thus RQ 2.

5.5.1 RQ 1: Evaluation of Feature Model and Sampling Algorithm for Scenario Generation

Considering RQ 1, we discuss each research question individually according to the experiment results and conclude them afterward. We discuss the influence of the FM and the sampling strategy on the mutation score (RQ 1.1). We evaluate the composition of a scenario suite the correlation to the mutation score (RQ 1.2). Subsequently, we discuss the relevance of single features on the mutation score (RQ 1.3).

RQ 1.1: Influence of Feature Model and Sampling Algorithm on Mutation Score

We expect, increasing variation diversity provides, under certain conditions, stronger scenario suites. Beyond that, we expect different strong scenarios suites if they are generated using different sampling algorithms.
Surprisingly, within our experiment (see Section 5.4.1), we see, that the sampling algorithms do not affect the mutation score of the scenario suites for FM 1 and FM 2. According to the mutation score, the seven sampling strategies provide equally strong scenario suites for FM 1 and FM 2. In contrast, we identify for FM 3 various mutation scores for scenario suites generated by various sampling algorithms. That means for FM 3, using different sampling strategies results in scenario suites that are differently strong according to the mutation score. For FM 3, we identify Chvatal ($t = 2$), Chvatal ($t = 3$), ICPL ($t = 3$), and IncLing generate scenario suites that are equally strong, while ICPL ($t = 1$) generates a scenario suite that is weaker within our experiments. The feature-wise sampling algorithms ICPL ($t = 1$) and Chvatal ($t = 1$) result in the weakest scenario suites.

https://doi.org/10.24355/dbbs.084-202102191146-0

Sampling algorithms extract configurations of a multi-variant system to represent the whole configuration space using restricted effort. The more complex the multi-variant system and the larger the configuration space, the greater the impact of various sampling strategies on the set of representing configurations. Considering the complexity of the FMs of our experiments, we identify, that the influence of the sampling algorithms on mutation score increases for FM 3 thus the FM with the highest complexity. We expect that FM 1 and FM 2 are not complex enough for scenario suite generation to obtain differences in the mutation score. According to support this proposition, we generate all valid configurations for FM 2. We get 39 configurations for FM 2. Comparing the number of all valid configurations to the number of generated scenarios per sampling algorithm in Table 5.1, we identify, that the sampling using pair-wise sampling algorithms ICPL ($t = 2$), Chvatal ($t = 2$), and IncLing result in two configurations less. We can represent the configuration space of FMs with little complexity as a whole and we assume that an expert knowledge-based scenario generation is manageable.

RQ 1.2: Influence of Single Scenarios on the Mutation Score

Considering the mutation score of single scenarios and the classification we introduce in Section 5.4.2, we identify, that one peak at a mutation score of $MS_M \approx 0,05$ and another one at $MS_M \approx 0,42$. We separate the classes by a threshold of $\theta_{MS} = 0,2$. Comparing the mutation score of single scenarios to the union of an overall scenario suite, we recognize, that the mutation score for each scenario is less than the overall scenario suite. We assume, that various scenarios kill various mutants thus the combination of scenarios within a scenario suite is relevant for the mutation score.

According to our observations of Section 5.4.1, we recognize an increasing number of scenarios according to the FM complexity. Comparing the number of scenarios for the three FMs, we see, that we create more scenarios for FM 3 except for feature-wise sampling strategies (see Table 5.1 and Figure A.3). For the sampling strategies ICPL ($t = 2$), ICPL ($t = 3$), Chvatal ($t = 2$), Chvatal ($t = 3$), and IncLing we identify an increasing number of scenarios. Beyond that, the ratio of both classes varies according to the FM. Across all sampling strategies, we identify a strong increase of scenarios in the class $MS \leq \theta_{MS}$ for FM 3 (see Table 5.4). However, an increasing number of scenarios means an increasing simulation effort due to a larger number of simulations. Considering the number of scenarios within the two classes according to the mutation score, we identify, that the ratio of scenarios resulting in a mutation score of $MS \leq \theta_{MS}$ increases with increasing complexity. As previously mentioned, we assume that the combination of scenarios within a scenario suite is relevant for the mutation score of the scenario suite. Nevertheless, we assume, that the scenarios resulting in a scenario mutation score $MS \leq \theta_{MS}$ are not as relevant as the scenarios of the class $MS > \theta_{MS}$. For this reason, we identify an increasing simulation effort without effect on the mutation score for scenario suites basing on complex FMs. However, in contrast to a less complex FM, a more complex FMs provide more variants aiming for a stronger model of the reality and test results that are more relevant.

https://doi.org/10.24355/dbbs.084-202102191146-0

RQ 1.3: Influence of Single Features on the Mutation Score

In Section 5.4.3, we present the relevance of single features on the mutation score. Considering the feature relevance of both, maneuver-features and time-invariant features, we do not recognize a single, optional child-feature that occurs in every scenario that kills a mutant. Beyond that, we do not see a significant difference between maneuver-features and time-invariant features. According to Section 5.4.2, there is no single scenario within a scenario suite that kills all mutants, anyway. As well as the combination of various scenarios is relevant to generate a strong scenario suite, we identify that the combination of features is relevant to create a strong scenario suite, too.

However, we identify more relevant feature categories than others. From a statistical point of view, trailer features do not contribute as much as features from the category *Environment wind*. Beyond that, we recognize that the relevance of maneuver features varies. We identify the maneuver that bases on the Euro NCAP Test scenarios, are noticeable more relevant than, for instance, *AEB_CrossingCarIntersection*. One reason for this could be that this maneuver is the only maneuver triggering a collision with another car, not a vulnerable road user such as the NCAP maneuvers do. Otherwise, we identify the maneuver *AEB_CrossingPedestrianBusStop* as the maneuver with the smallest relevance within our experiments. The maneuver *AEB_CrossingPedestrianBusStop* triggers the AEB VRU such as the NCAP maneuver leading to a higher relevance.

Subsequently, we can not identify single features that lead to a stronger scenario suite according to the mutation score but we recognize the trend of various categories to be more relevant than others. For this reason, we can modify the FMs aiming for stronger scenario suites. However, these results also confirm our expectation, that not single features or scenarios but the combination of features and scenarios are relevant for the strength of a scenario suite.

RQ 1: Conclusion

According to evaluate the research question RQ 1, we combine the findings we discussed for the research questions RQ 1.1, RQ 1.2, and RQ 1.3. We identify a correlation between the mutation score and the sampling strategy, as well as between the mutation score and the FM.

Considering the correlation between FM and mutation score, we recognize that only FM 3 leads to a varying mutation score for different sampling strategies. We assume a correlation between the complexity of the FM and the mutation score. For our experiment, we set up the FMs manually with a different complexity. We calculate a mutation score of $MS \approx 0,504$ thus we kill approximately 50% of the mutants within the mutant test suite. A mutant killing rate of 50% is as good as flipping a coin. However, we identify varying mutation scores for different sampling strategies if we generate the scenario suites based on the most complex FM 3.

We investigate the composition of the scenarios suites that we generate using various sampling strategies on the three FMs. We identify a distribution, of the scenarios within a scenario suite, into two classes. We rate the scenarios according to the

https://doi.org/10.24355/dbbs.084-202102191146-0

mutation score that we calculate for each scenario. Some scenarios seem to be more relevant than others. The main finding is that there is no single scenario that kills all mutants. Compared with the mutation score of the union of all scenarios within a scenario suite, each single scenario leads to a smaller mutation score. We identify the combination of the various scenarios to be relevant for the strength of a scenario suite.

The investigation of the relevance of single scenarios lead to a similar insight. We do not identify single features to be excessively relevant to kill mutants, neither maneuver-features nor time-invariant features. We do identify categories that are more relevant than others, such as *Environment Wind* is more relevant on average than *Trailer*. However, we deduce that the combination of various features to be more relevant than the appearance of a single feature.

Conclusively, we identify the combination of features resulting in a scenario, as well as the combination of scenarios resulting in a scenario suite, influencing the mutation score. Various sampling strategies lead to various mutation scores, indicating the strength of the test environment. Beyond that, we identify that the influence, we previously mentioned, only appears for the FM of the highest complexity within our experiments. We assume a correlation of the influence between mutation score and sampling strategies depending on the FM and its complexity.

5.5.2 RQ 2: Mutation Score as a Metric to rate Scenario Suites

For an evaluation of RQ 2, we discuss our observations regarding the correlation between mutant test suite and sampling strategy according to kill specific mutants (RQ 2.1). Subsequently, we conclude our findings regarding the mutation score as a metric to rate scenarios suites afterward.

RQ 2.1: Impact of Scenario Suites on Mutant Killing

In Section 5.4.4 we calculate the detection rate of single mutants according to various scenario suites. Considering the detection rate of single mutants we recognize, that various mutants are different hard to kill. Thus a rating of single mutants or the mutant test suite according to the mutation score is possible. We also identify that scenario suites generated using various sampling algorithms lead to various detection rates for the same mutant. Thus the rating of the mutants depends on the used scenario suite. If we fix the mutant test suite, we get to know, which scenario suite detects which mutants. Beyond that, the mutant detection rate indicates, how sure a scenario suite detects a mutant. Within our experiments for FM 3, we identify that the mutants *Accel_AEB_mut_44*, *Accel_AEB_mut_71* are not detected by any scenario of the scenario suites generated using Chvatal ($t = 1$), ICPL ($t = 1$) and ICPL ($t = 2$). Beyond that, *Accel_AEB_mut_6* is only detected by ICPL ($t = 2$). These results correlate with our findings of RQ 1.3.

RQ 2: Conclusion

According to evaluate the research question RQ 2, we combine the findings we previously mentioned. The mutation score represents the ratio of killed mutants

https://doi.org/10.24355/dbbs.084-202102191146-0

regarding the overall number of known mutants thus the mutation score evaluates the quality of the testing. This means we create mutants resulting in ground truth data and evaluate the quality of the testing regarding this ground truth data. If we only change one parameter within the test process chain, we identify the correlation between the changed parameter and the output, thus the mutation score. In Section 5.4.1 we change the FM and the sampling strategy independently. We identify a correlation between FM and mutation score, as well as for FM 3 a correlation between sampling algorithm and mutation score. We use FM as well as sampling algorithms to generate scenario suites. Thus we identify a correlation between the mutation score and the scenario suite. According to the fact that the mutation score evaluates the quality of a testing process and we only vary the scenario suites within our testing process, we deduce a correlation between the mutation score and the strength of a scenario suite. Thus we identify the mutation score as a metric to rate the strength of scenario suites.

Concurrently, we identify limits of the mutation score rating the strength of scenario suites. Our findings for RQ 2.1 show, that there are mutants that are hard to kill as well as there are mutants that are easy to kill. Beyond that, we identify that the mutants are different hard to detect for various scenario suites. These differences result in varying mutation scores for various scenario suites. Beyond that, we identify varying mutant detection rate for various mutants using the same scenario suite. Within our evaluation, we focus on one mutant test suite to generate comparable results. Nevertheless, we assume a correlation between the mutation score and the mutant test suite.

We expect to increase the mutation score by eliminating equivalent mutants in the mutant test suite. The AEB model of the subject system, for instance, originally implements staged braking using two stages. We do not use the first braking steps within this thesis by determinating the output (see Figure 5.4). Nevertheless, SIMULTATE [PRWN16] and MTAF [Mev19] implements mutation operators within this area resulting in equivalent mutants according to Grün et al. [GSZ09]. However, we have to define a mutant test suite or specific, comparable parameters of the mutant test suite. We need comparable mutant test suites to use the mutation score to evaluate the strength of various scenario suites traceably. There are approaches to rank mutants mainly motivated due to reduce the number of mutants [GSZ09, Hus08]. In contrast, we aim for an evaluation of a given mutant test suite regardless of the composition.

5.6 Threats to Validity

Within this section, we investigate the validity of our results and discuss potential threats regarding our study. We structure the discussion into the consideration of the internal validity, external validity, and construct validity.

5.6.1 Internal Validity

Evaluation Bias: An impact on the results due to an invalid evaluation is conceivable, for instance, due to a bug in the calculation scripts. We counter an invalid evaluation due to a conscientious implementation. Beyond that, we think about

https://doi.org/10.24355/dbbs.084-202102191146-0

possible results before the evaluation and compare the computed results to our expectations. We also question the results in iterative discussions and compare various findings to each other.

Experimenter Bias: We work on a process chain to generate scenario suites automatically. Nevertheless, we have to extract the configurations from FeatureIDE, start up the transformation tool and the simulation tool manually. We execute the work steps with a high degree of certainty. Beyond that, we support the user with a high degree of automation and scripts to create, for instance, folder structures, give instructions, or verify data.

Bias of the Feature Model Creation: We do not use a common FM from practice. Thus we create a FM based on expert-knowledge as an input for the scenario generation process chain. For this purpose, we deduce knowledge from the documentation of the simulation tool [IPGb, IPGc, IPGd] and further literature [VWR18, UMR+15, Sch17]. The FM may have inaccuracies or bugs leading to falsified results. We implement the FM conscientiously within an iterative process and evaluate the implementation on-going due to simulations. We use the IPGMovie tool to evaluate the FM after adding new features. Beyond that, the CarMaker simulation tool notifies if it is not possible to execute a simulation due to an incorrect scenario. We trace the incorrect scenario back to the FM or the related databases to transform configurations into executable scenarios. Thus, we reduce possible inaccuracies or bugs within our FM to a minimum.
According to the maneuver feature integration concept, the maneuver-features build an essential part of the FM. We implement the maneuver-features according to the provided scenarios of IPG Automotive GmbH [IPGa]. These scenarios are provided in the form of executable InfoFiles for CarMaker and designed to trigger the AEB of a vehicle. We modify the scenarios into maneuver templates and separate time-invariant elements. Thereby, we might implement bugs into the maneuver template or the time-invariant features. However, we aim for a sampling based composition of a scenario thus we do not need to reconstruct the origin scenario without bugs. But it is important to generate executable scenarios due to sampling thus we evaluate each feature on a random basis using CarMaker and IPGMovie.

Bias of the Mutant Test Suite: We create a mutant test suite using the tools MTAF [Mev19] and SIMULTATE [PRWN16]. Thereby, we implement mutation operators inspired by [Mev19] resulting in a 150% mutant model. We use one mutant test suite due to extract first order mutants. In Section 5.4.4, we identify different mutant detection rates for different mutants within the mutant test suite. Thus we conclude that using various mutant test suites could affect the mutation scores. We use the same mutant test suite for all simulations we perform within this thesis aiming for comparable results. However, due to a temporal limit, we do not perform the same tests using various mutant test suites to evaluate the impact of the mutant test suites or single mutation operators on the subject system.

https://doi.org/10.24355/dbbs.084-202102191146-0

Bias of Sensor Influence: We implement various time-invariant features into our FMs. For instance, we define the kind of crossing object. The sampling strategies lead to the composition within a resulting scenario. The kind of the crossing object is essential for the sensors we use as input for the ADAS model. Various objects may result in different sensor outputs. Within our study, we always use the same sensor provided by IPG Automotive GmbH [IPGd, IPGc]. Using various sensor model may lead to different results. However, we do not focus on the sensor data processing, but on the control algorithms. Thus we use the same sensor within the overall study. This leads to comparable results within our study, if we investigate modifications within the control algorithms. We implement the control algorithms within our ADAS model.

5.6.2 External Validity

Generalizability accross Situations: We implement a FM that is adjusted to represent scenarios that triggers the AEB VRU of a motor vehicle. We also add one maneuver to trigger the AEB due to a pending collision with another vehicle. Nevertheless, the FM is specialized to investigate an ADAS, such as an AEB, as well as our findings according to the sampling strategies. Our focus is the evaluation of whether there are differences between various sampling strategies for automated scenario generation to support ADAS/AD testing. Nevertheless, we implemented our process chain generic. Thus further studies are thinkable to evaluate the quality of sampling strategies more generally within this topic.

Generalizability Complexity Evaluation: In Section 5.4 and Section 5.5, we combine the complexity of the FM with the mutation scores of various scenarios suites. Within our experiments, we recognize significant differences for FM 3. However, we used the same process chain leading to reproducible results. The differences within the FM come with a higher number of maneuver features and time-invariant features. In Section 5.4.3, we investigate the influence of single features on the mutation score for FM 3. We do not identify single features that are of significant relevance. Beyond that, we investigate the coverage of the sampling strategies according to all valid configurations in the form of configuration count. These arguments suggest to conclude a correlation between FM complexity and mutation score for the freely chosen subject system. We expect similar relations for further subject systems.

Generalizability accross ADAS: We perform the tests using an ADAS in the form of an AEB. Thereby, we do not use a common system from practice, but we introduce a Simulink model that we implement inspired by [Arc18]. There might be bugs within the AEB model we use as the origin for the mutation testing. To counter these bugs, we implement the AEB model conscientiously within an iterative process. We evaluate the function of the AEB model due to on-going simulations. Beyond that, we simulate the origin model for each scenario we use in our evaluation resulting in the expected outputs for the test cases using mutant models as SUT.

We implement a single ADAS model to evaluate whether the FM or the sampling strategies have an impact on the mutation score. Within this thesis, we prove that

https://doi.org/10.24355/dbbs.084-202102191146-0

there is an impact on our subject system, thus for at least this ADAS. We do not investigate further ADAS or quantize the results due to a temporal limit, but provide a Simulink interface to perform an evaluation using further ADAS.

5.6.3 Construct Validity

Bias in Experimental Design: We use the mutation score to evaluate the strength of the scenario suite. For this purpose, we implement the system specification, in the form of a collision detection, into a SEC. Thus, we investigate the influence of a mutated AEB model on the function of the overall vehicle in accordance with its specification. Next to the SUT (original ADAS or mutant), there might be an influence due to, for instance, further ADAS. We deactivate further ADAS and use the same vehicle model for all simulations to keep the influences as small as possible.

https://doi.org/10.24355/dbbs.084-202102191146-0

6. Related Work

In this chapter, we mention related work. We present similar work and differentiate it from this thesis. We handle related work regarding scenario-based testing, sample-based scenario generation, and mutation testing in the automotive context.

Scenario-based testing

Scenario-based testing is a suitable approach for verification and validation of AD-functions. There are various research activities in and around this field. For instance, Menzel et al. [MBM18] developed an essential definition of the term scenario with different abstraction levels in the ADAS/AD context. In addition to cost and time reduction, Bagschik et al. [BMKM18] also mention traceability as the main advantages of automated scenario generation.
Bagschik et al. [BMKM18] work on the generation of scenarios. They develop an ontology-based approach to create a description of a scene. Thereby, they implement an ontology that represents the knowledge of a scene inspired by a layer model of Schuldt [Sch17]. The knowledge for the ontology bases on the guidelines for german motorways. Based on a start scene that Bagschik et al. [BMKM18] generate, they derive possible end scenes and relations between them, resulting in a scenario. Bagschik et al. [BMKM18] export the scenarios within a technical scenario graph as well as a HTML-Visualisation. They evaluate the generated scenarios due to a verification according to the input graph. This means they investigate the correctness of the scenario generation. Bagschik et al. [BMKM18] do not rate the resulting scenarios in combination with the detection of bugs within an ADAS/AD. Beyond that, they do not cluster the resulting scenarios into scenario suites.
Menzel et al. [MBI+19] investigate different approaches to generate scenarios. Therefore, the authors give an overview of publications dealing with data-driven and knowledge-driven approaches. Concurrently, they introduce a concept that can transform keyword-based scenarios into common simulation formats automatically, also mentioned in [MBI+18]. This approach works straightforward and bases on an ontology-based scenario description. For this purpose, Menzel et al. [MBI+19] transfer the scenarios from a semantic scenario representation into the data formats OpenDRIVE and OpenSCENARIO used by the simulation tool Virtual Test

https://doi.org/10.24355/dbbs.084-202102191146-0

Drive. Menzel et al. [MBI+19] evaluate the resulting scenarios according to a correct transformation. They do not rate the resulting scenarios in combination with the detection of bugs within an ADAS/AD. They also do not cluster the resulting scenarios into scenario suites.

Sample-based scenario generation

Vogelsang et al. [VWR18] consider the simulation tool as a Software Product Line (SPL). For this approach they investigate the simulation tool itself. They understand properties and possibilities of the simulation tool as a FM with single attributed features and dependencies between them. Consequently, a configuration builds the input for a scenario generator from which a scenario results. In a second step, they investigate the variation of attributed features, so-called agents. As a result, Vogelsang et al. [VWR18] identify the engineer who chooses a subset of features for simulation as a significant influence on the result but do not discuss this in detail. Inspired by the work of Vogelsang et al. [VWR18], we investigate the approach to understand a simulation tool as a configurable system. In contrast, we do not subdivide into domain and application engineering like Vogelsang et al. [VWR18] do in the context of software product lines. We focus on various configurations and sampling strategies indeed. However, same as with the work of Vogelsang et al. [VWR18], the process chain that is developed within this thesis bases on a FM. We derive the FM from the CarMaker simulation environment and represents possible scenarios. Later we need to transfer the configurations in to executable scenarios for CarMaker.

Mutation testing in automotive context

Mevenkamp [Mev19] applies mutation testing to evaluate the quality of a test set according to verify or validate autonomous Vehicles. The mutation score builds a metric for the evaluation. Beyond that, Mevenkamp [Mev19] investigates the influence of *scenario fuzzing* on the mutation score. Fuzzing is a test method that is used especially for software testing. Fuzzing brings a certain random factor into test cases generated by experts, thus reducing the subjective, human influence, and making the results more robust. For this purpose, Mevenkamp [Mev19] selects an existing scenario and varies it using specially defined *fuzzing parameters*. In doing so, Mevenkamp does not generate scenarios due to sampling but varies parameters such as the traveling speed of road users within a given scenario. However, Mevenkamp creates a tool named *MTAF* that fuzzes scenarios, build mutations of a Simulink model for CarMaker, and tests them. Beyond that, Mevenkamp uses a SEC as a kill criterion for the mutants. Within this thesis, we create the mutants inspired by Mevenkamp and we also implement a SEC in the form of a Simulink model. In contrast, we focus on various sampling strategies scenarios instead of fuzzing parameters and we do not implement the SEC directly into the ADAS model. Beyond that, we use a different interface to connect the Simulink ADAS model to CarMaker than Mevenkamp.

https://doi.org/10.24355/dbbs.084-202102191146-0

7. Conclusion

This thesis contributes to scenario generation and evaluation for scenario-based testing. We introduce a process chain to generate scenarios automatically. This process chain bases on sampling on a FM that is derived from a simulation tool. For this purpose, we suggest three concepts to represent scenarios using a FM. These three concepts differ in the representation of scenario elements that vary over time. Two approaches deal with the combination of multiple static scenes. The third approach implements a combination of maneuver features and time-invariant features within one FM. However, we implement the maneuver integration approach as a proof of concept. We sample configurations from the FM and transform them into executable scenarios for a simulation environment. This results in a set of scenarios, which we define as a scenario suite. We generate scenario suites using various t-wise sampling strategies.

We suggest to use mutation testing to evaluate scenario suites. As a metric, we use the mutation score. For this purpose, we create a 150% mutant model of an ADAS by inserting mutation operators. We extract mutants from the 150% mutant model resulting in a set of mutants which we define as mutant test suite. As a kill criterion, we use a SEC that implements the specifications of the overall vehicle.

We evaluate the impact of three FM with different complexity, and common sampling algorithms on the mutation score using an AEB as subject system. We identify that the sampling strategies have an impact on the mutation score for FM 3, the FM that implements the highest complexity. Within our study, the 3-wise and pair-wise sampling strategies lead to stronger scenario suites than feature-wise sampling. However, we identify no impact of the sampling strategy in combination with FM 1 and FM 2. All scenario suites for FM 1 and FM 2 lead to a mutation score of $MS \approx 0,5$ thus we detect only approximately 50% of false negatives. Using FM 3 in combination with the feature-wise sampling algorithm ICPL ($t = 1$), Chvatal ($t = 1$) as well as the pair-wise sampling algorithm ICPL ($t = 2$), we kill less than 50% of the mutants. A killing rate of 50% is as good as flipping a coin. We expect to increase the mutation score by analyzing the mutant test suite aiming to eliminate equivalent mutants. However, we conclude, that the FM in combination with the

https://doi.org/10.24355/dbbs.084-202102191146-0

sampling algorithm influence the mutation score thus the strength of the scenario suite.

Beyond that, we investigate single scenarios within a scenario suite and single features within a scenario. Thereby, we identify, that no single scenario or feature leads to the overall mutation score of a scenario suite. But we identify feature categories that are more relevant than others. We conclude, that the combination of different scenarios, as well as different features, are relevant to generate a strong scenario suite. This finding maps with the fact, that 3-wise sampling algorithms lead to stronger scenario suites for FM 3 than feature-wise sampling algorithms. In general, we identify the FM as a suitable method to represent scenarios.

We investigate the impact of the mutant test suite to evaluate, whether the mutation score is a practical metric to rate scenario suites. We identify an impact of the scenario suites on the mutation score thus the mutation score evaluates the scenario suites. We also identify that the detection rate of different mutants varies for different scenarios suites. Some scenario suites detect specific mutants, others do not. Beyond that, different scenario suites detect specific mutants with varying reliability. In conclusion, we identify that the we can use the mutation score to evaluate the strength of scenario suites. Nevertheless, there is a limit due to the impact of the mutant test suite. For a comparative evaluation of scenario suites using the mutation score, we need to define parameters that describe a mutant test suite.

https://doi.org/10.24355/dbbs.084-202102191146-0

8. Future Work

In this chapter, we depict possibilities for future work that directly follows the findings of this thesis. According to Figure 5.1, we identify four essential parameters influencing the mutation score within our process chain. We identify that the FM and the sampling algorithms have an impact on the generation of a scenario suite. We investigate three FMs in combination with seven sampling algorithms according to our subject system. The FMs are designed to generate scenarios that trigger an AEB within a motor vehicle. Considering increased generalizability, we suggest to create a FM representing all possibilities a simulation tool provides to create scenarios. In Section 3.1.2, we suggest three concepts to implement the FM. A FM, that represents all possibilities of a simulation tool leads to generate scenario suites for further ADAS/AD. According to reduce the simulation effort, we suggest to slice the overall FM specialized to verify and validate single ADAS/AD. For this purpose, we suggest to evaluate the relevance of single features and feature categories according to our investigation presented in Section 5.4.3. Beyond that, we suggest to investigate further sampling strategies. We also suggest to implement and investigate the multiple scene creation concepts for the FM-design. These concepts may require modified sampling strategies.

Considering the impact of the mutant test suite, we investigate in Section 5.4.4, we suggest to evaluate the impact of various mutant test suites. This includes an investigation of single mutants as well as the composition of various mutants within a mutant test suite. We suggest to define parameters to describe the mutant test suite according to generate comparable mutation scores for an evaluation of the various scenario suites.

Finally, we suggest to investigate the influence of the SEC on the mutation score. We define a SEC as a kill criterion for the mutation testing. The SEC implements the specification of the overall vehicle. Within this thesis, we use a generalized definition. We define that there must not be a collision. Various definitions may lead to different mutation scores. Beyond that, we suggest to use the SEC to develop a rule-based specification of an ADAS that uses, for instance, artificial intelligence.

https://doi.org/10.24355/dbbs.084-202102191146-0

For this purpose, we can use our process chain with unchanged input parameters and develop a SEC according to an optimum of the mutation score.

https://doi.org/10.24355/dbbs.084-202102191146-0

A. Appendix

https://doi.org/10.24355/dbbs.084-202102191146-0

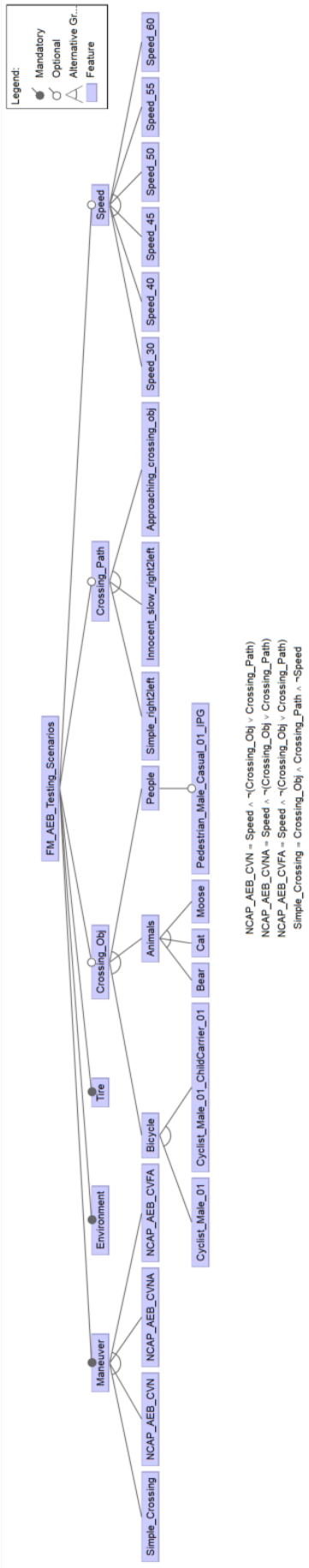


Figure A.1: FM 2 to generate scenario suites; complementary to Section 5.4.1

https://doi.org/10.24355/dbbs.084-202102191146-0

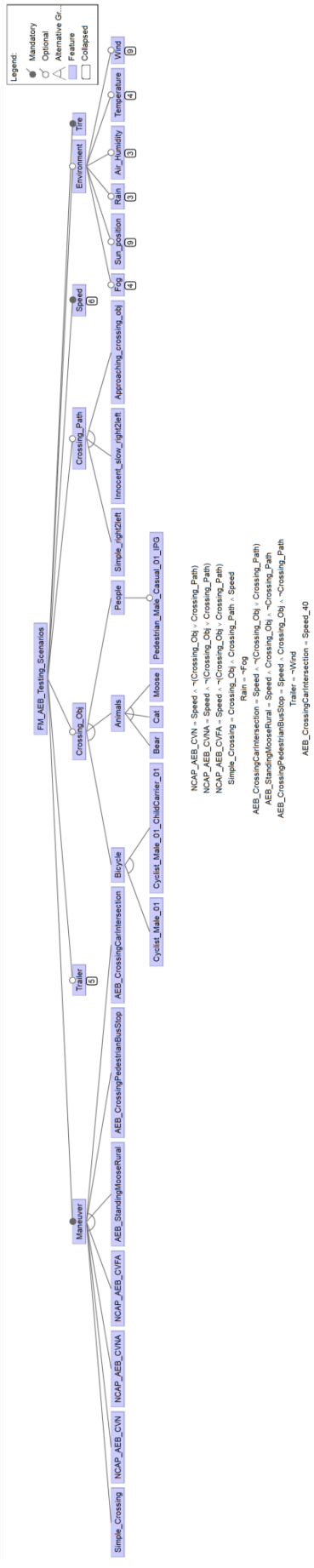


Figure A.2: FM 3 to generate scenario suites; complementary to Section 5.4.1

https://doi.org/10.24355/dbbs.084-202102191146-0

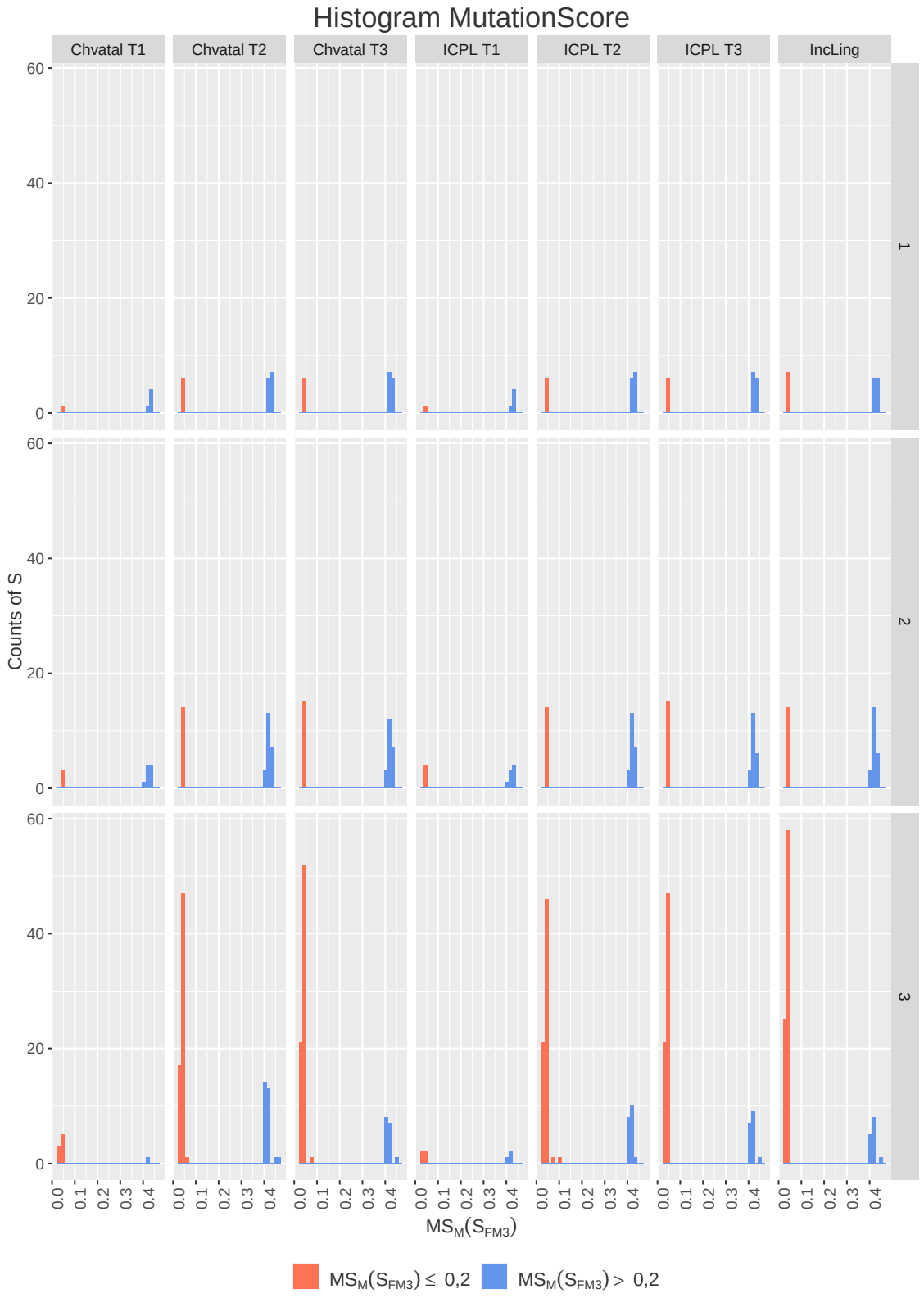


Figure A.3: Distribution of scenario-based MS per FM and sampling algorithm; complementary to Section 5.4.2 (rows: FM, columns: scenario suite)

https://doi.org/10.24355/dbbs.084-202102191146-0

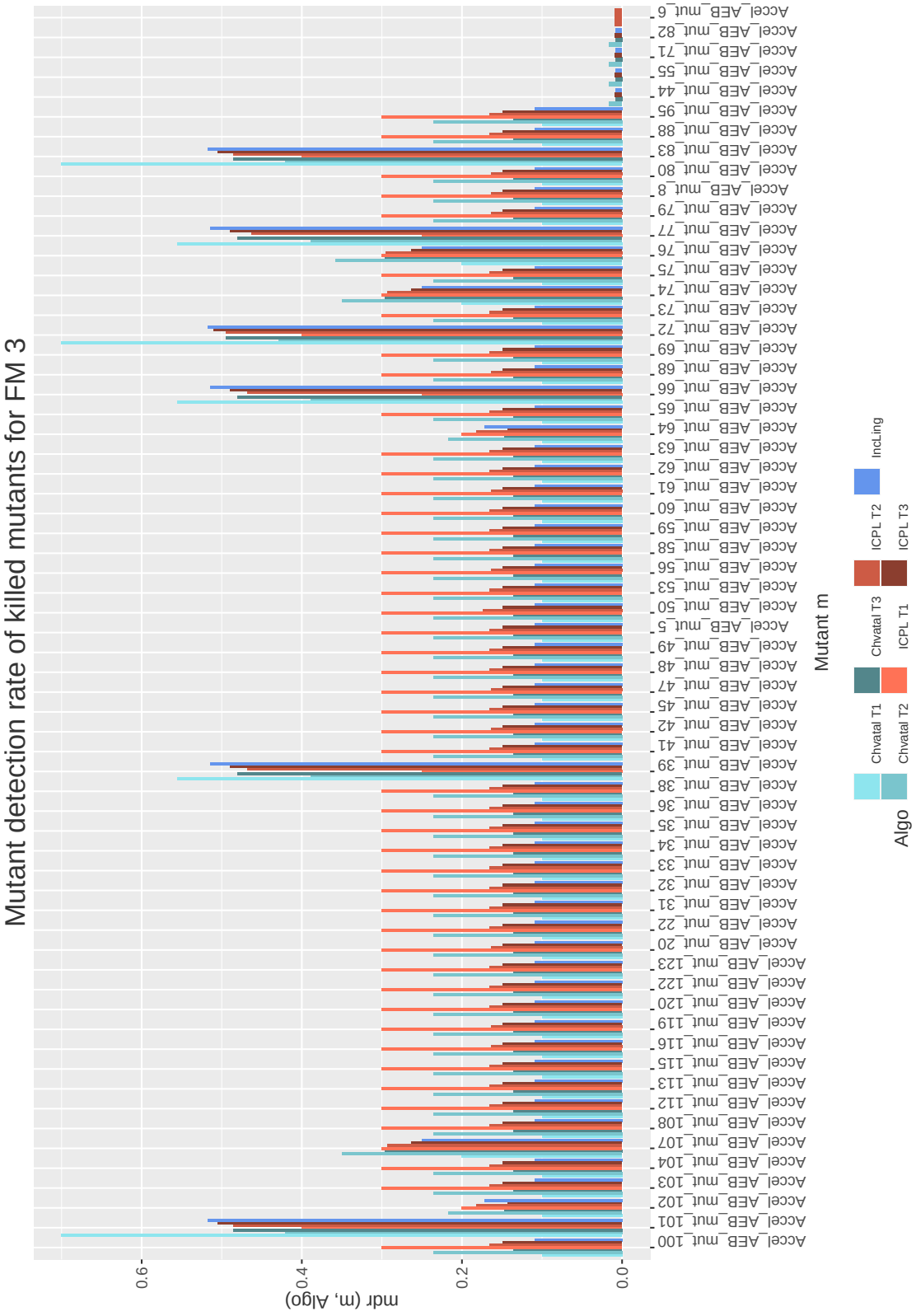


Figure A.4: Mutant detection rate for FM 3; complementary to Section 5.4.4

https://doi.org/10.24355/dbbs.084-202102191146-0

https://doi.org/10.24355/dbbs.084-202102191146-0

Bibliography

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer-Verlag GmbH, 2013. (cited on Page 5, 6, 7, 27, and 28)

[Abt13] Dietmar Abts. *Grundkurs JAVA : von den Grundlagen bis zu Datenbank- und Netzanwendungen*. Springer Vieweg, Wiesbaden, 2013. (cited on Page 33)

[AHKT+16] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. Incling: efficient product-line testing using incremental pairwise sampling. *ACM SIGPLAN Notices*, 52(3):144–155, 2016. (cited on Page 7, 8, 17, 29, and 50)

[AHMK+16] Mustafa Al-Hajjaji, Jens Meinicke, Sebastian Krieter, Reimar Schröter, Thomas Thüm, Thomas Leich, and Gunter Saake. Tool demo: testing configurable systems with featureide. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 173–177, 2016. (cited on Page 28)

[AHTL+19] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. Effective product-line testing using similarity-based product prioritization. *Software & Systems Modeling*, 18(1):499–521, 2019. (cited on Page 17)

[AJ80] Allen Troy Acree Jr. On mutation. Technical report, GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION AND COMPUTER SCIENCE, 1980. (cited on Page 9)

[Arc18] Alberto Arcidiacono. *ADAS virtual validation: ACC and AEB case study with IPG CarMaker*. PhD thesis, Politecnico di Torino, 2018. (cited on Page vii, viii, 37, 50, 51, 52, and 67)

[ASAa] ASAM e.V. *ASAM OpenSCENARIO v 1.0.0*. (cited on Page 30)

[ASAb] ASAM e.V. *OpenCRG Version 1.2.0*. (cited on Page 30)

[ASAc] ASAM e.V. *OpenDRIVE 1.6*. (cited on Page 30)

https://doi.org/10.24355/dbbs.084-202102191146-0

[AWS14] Harald Altinger, Franz Wotawa, and Markus Schurius. Testing methods used in the automotive industry: Results from a survey. In *Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing*, pages 1–6, 2014. (cited on Page 1, 9, 20, and 30)

[B+12] Nguyen Thanh Binh et al. Mutation operators for simulink models. In *2012 Fourth International Conference on Knowledge and Systems Engineering*, pages 54–59. IEEE, 2012. (cited on Page 20)

[BA82] Timothy A Budd and Dana Angluin. Two notions of correctness and their relation to testing. *Acta informatica*, 18(1):31–45, 1982. (cited on Page 9)

[BMKM18] G Bagschik, T Menzel, C Körner, and M Maurer. Wissensbasierte szenariengenerierung für betriebsszenarien auf deutschen autobahnen. In *Workshop Fahrerassistenzsysteme und automatisiertes Fahren. Bd*, volume 12, 2018. (cited on Page 12 and 69)

[Boa19] International Software Testing Qualifications Board. Standard glossary of terms used in software testing version 3.2, October 2019. (cited on Page 1)

[Boc09] Thomas Bock. Bewertung von fahrerassistenzsystemen mittels der vehicle in the loop-simulation. In *Handbuch Fahrerassistenzsysteme*, pages 76–83. Springer, 2009. (cited on Page 11)

[BPPK09] Goetz Botterweck, Andreas Pleuss, Andreas Polzer, and Stefan Kowalewski. Towards feature-driven planning of product-line evolution. In *Proceedings of the First International Workshop on Feature-Oriented Software Development*, pages 109–116, 2009. (cited on Page 27)

[Cap] Captain feature. https://sourceforge.net/projects/captainfeature/. [accessed 2021-01-04]. (cited on Page 27)

[CDFP97] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The aetg system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997. (cited on Page 7)

[CDS07] Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 129–139, 2007. (cited on Page 7)

[CDS08] Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering*, 34(5):633–650, 2008. (cited on Page 7)

https://doi.org/10.24355/dbbs.084-202102191146-0

[Chv79] Vasek Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3):233–235, 1979. (cited on Page 7 and 8)

[DLS78] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978. (cited on Page 8)

[DRC+17] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator. *arXiv preprint arXiv:1711.03938*, 2017. (cited on Page 41)

[Eur15a] Euro, NCAP. Test protocol-aeb vru systems. https://cdn.euroncap.com/media/21509/euro-ncap-aeb-vru-test-protocol-v101.pdf, 2015. [accessed 2020-12-13]. (cited on Page 49)

[Eur15b] European New Car Assessment Programme. Euro ncap rating review 2015 - report from the ratings group. https://cdn.euroncap.com/media/16470/ratings-group-report-2015-version-10-with-appendix.pdf, 2015. [accessed 2020-12-13]. (cited on Page 50)

[Gey13] Sebastian Geyer. *Entwicklung und Evaluierung eines kooperativen Interaktionskonzepts an Entscheidungspunkten für die teilautomatisierte, manöverbasierte Fahrzeugführung*. Number 770. VDI, 2013. (cited on Page 22)

[GGS09] Christhard Gelau, Tom Michael Gasser, and Andre Seeck. Fahrerassistenz und verkehrssicherheit. In *Handbuch Fahrerassistenzsysteme*, pages 24–32. Springer, 2009. (cited on Page 1)

[GSZ09] Bernhard JM Grün, David Schuler, and Andreas Zeller. The impact of equivalent mutants. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 192–199. IEEE, 2009. (cited on Page 9, 25, and 65)

[Hei11] Cornelia Heinisch. *Java als erste Programmiersprache : vom Einsteiger zum Profi ; [Java 6*. Vieweg + Teubner, Wiesbaden, 2011. (cited on Page 33)

[Hus08] Shamaila Hussain. Mutation clustering. *Ms. Th., Kings College London, Strand, London*, 2008. (cited on Page 9 and 65)

[IPGa] IPG Automotive GmbH. Carmaker. https://ipg-automotive.com/de/produkte-services/simulation-software/carmaker/. [accessed 2020-12-08]. (cited on Page vii, viii, 3, 12, 13, 41, 51, 52, and 66)

[IPGb] IPG Automotive Group. *Programmer's Guide*. (cited on Page 13, 29, 30, 37, 38, 40, 42, 49, 51, and 66)

[IPGc] IPG Automotive Group. *Reference Manual*. (cited on Page 13, 29, 38, 41, 49, 51, 66, and 67)

https://doi.org/10.24355/dbbs.084-202102191146-0

[IPGd] IPG Automotive Group. *User's Guide*. (cited on Page 12, 13, 29, 30, 42, 44, 49, 66, and 67)

[Jac12] Paul Jaccard. The distribution of the flora in the alpine zone. 1. *New phytologist*, 11(2):37–50, 1912. (cited on Page 17)

[JH10] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010. (cited on Page 8, 9, 20, 21, and 24)

[JHF11] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. Properties of realistic feature models make combinatorial testing of product lines feasible. In *International Conference on Model Driven Engineering Languages and Systems*, pages 638–652. Springer, 2011. (cited on Page 7, 8, 17, 20, 29, and 50)

[JHF12] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 46–55, 2012. (cited on Page 7, 8, 17, 20, 29, and 50)

[JS16] Thomas Zurawka Jörg Schäuffele. *Automotive Software Engineering*. Gabler, Betriebswirt.-Vlg, 2016. (cited on Page 1, 20, and 23)

[KWG04] D Richard Kuhn, Dolores R Wallace, and Albert M Gallo. Software fault interactions and implications for software testing. *IEEE transactions on software engineering*, 30(6):418–421, 2004. (cited on Page 7)

[MBI+18] T Menzel, G Bagschik, L Isensee, A Schomburg, and M Maurer. Detaillierung einer stichwortbasierten szenariobeschreibung für die durchführung in der simulation am beispiel von szenarien auf deutschen autobahnen-english title: Detailing a keyword based scenario description for execution in a simulation environment using the example of scenarios on german highways. In *Workshop Fahrerassistenzsysteme und automatisiertes Fahren*, volume 12, pages 15–26, 2018. (cited on Page 69)

[MBI+19] Till Menzel, Gerrit Bagschik, Leon Isensee, Andre Schomburg, and Markus Maurer. From functional to logical scenarios: Detailing a keyword-based scenario description for execution in a simulation environment. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 2383–2390. IEEE, 2019. (cited on Page 12, 69, and 70)

[MBM18] Till Menzel, Gerrit Bagschik, and Markus Maurer. Scenarios for development, test and validation of automated vehicles. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 1821–1827. IEEE, 2018. (cited on Page 1, 12, and 69)

https://doi.org/10.24355/dbbs.084-202102191146-0

[Mev19] Phillipp Mevenkamp. Mutation testing for autonomous vehicle. Master's thesis, Technical University of Braunschweig, 2019. (cited on Page vii, ix, 9, 20, 21, 22, 23, 36, 37, 40, 51, 52, 53, 65, 66, and 70)

[MSY] Msys. http://www.mingw.org/wiki/MSYS. [accessed 2020-12-04]. (cited on Page 40)

[NES17] Michael Nieke, Gil Engel, and Christoph Seidl. Darwinspl: an integrated tool suite for modeling evolving context-aware software product lines. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems*, pages 92–99, 2017. (cited on Page 27)

[OU01] A Jefferson Offutt and Roland H Untch. Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*, pages 34–44. Springer, 2001. (cited on Page vii and 10)

[Pet18] Tobias Pett. *Stability of Product Sampling under Product-Line Evolution*. PhD thesis, Master's thesis. TU Braunschweig, Germany, 2018. (cited on Page 17)

[PRWN16] Ingo Pill, Ivan Rubil, Franz Wotawa, and Mihai Nica. Simultate: A toolset for fault injection and mutation testing of simulink models. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 168–173. IEEE, 2016. (cited on Page vii, ix, 9, 20, 36, 51, 53, 65, and 66)

[pur] pure-systems GmbH. *pure::variants User's Guide*. (cited on Page 28)

[R C20] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2020. (cited on Page 44)

[RAAK12] S Rauch, M Aeberhard, M Ardelt, and N Kämpchen. Autonomes fahren auf der autobahn–eine potentialstudie für zukünftige fahrerassistenzsysteme. In *5. Tagung Fahrerassistenz*, 2012. (cited on Page 22)

[Res16] Andreas Reschka. *Safety Concept for Autonomous Vehicles*, pages 473–496. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. (cited on Page 22)

[Sch17] Fabian Schuldt. *Ein Beitrag für den methodischen Test von automatisierten Fahrfunktionen mit Hilfe von virtuellen Umgebungen*. PhD thesis, 2017. (cited on Page 1, 11, 12, 66, and 69)

[Sin11] Yogesh Singh. Structural testing. In *Software Testing*, pages 165–229. Cambridge University Press, 2011. (cited on Page 2, 20, 21, 23, and 25)

[SRC+12] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and

https://doi.org/10.24355/dbbs.084-202102191146-0

Karina Villela. Software diversity: state of the art and perspectives, 2012. (cited on Page 21)

[TKB+14] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014. (cited on Page 27 and 28)

[UMR+15] Simon Ulbrich, Till Menzel, Andreas Reschka, Fabian Schuldt, and Markus Maurer. Defining and substantiating the terms scene, situation, and scenario for automated driving. In *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, pages 982–988. IEEE, 2015. (cited on Page 1, 12, 16, 17, 23, and 66)

[VIRa] MSC.Software GmbH VIRES Simulationstechnologie GmbH. Vires virtual test drive (vtd) data sheet. https://media.mscsoftware.com/cdn/farfuture/-cvEJfmu1qc-Lzpt-F1gPoLbmrRjnh70prNZVYxXv2M/mtime:1603316972/sites/default/files/hexagon_mi_vires_datasheet-vtd_a4_web.pdf. [accessed 2021-01-04]. (cited on Page 30)

[VIRb] MSC.Software GmbH VIRES Simulationstechnologie GmbH. Virtual test drive. https://www.mscsoftware.com/de/virtual-test-drive. [accessed 2020-12-08]. (cited on Page 41)

[VWR18] Andreas Vogelsang, Massi Wakeli, and Stefan Rulewitz. Feature-oriented configuration of simulation scenarios. (unpublished), 2018. (cited on Page 2, 20, 66, and 70)

[WW16] Walther Wachenfeld and Hermann Winner. The release of autonomous vehicles. In *Autonomous driving*, pages 425–449. Springer, 2016. (cited on Page 1 and 11)

[WWP+14] Alexander Weitzel, Hermann Winner, Cao Peng, Sebastian Geyer, Felix Lotz, and Mohsen Sefati. Absicherungsstrategien für fahrerassistenzsysteme mit umfeldwahrnehmung. 2014. (cited on Page 11)

[YCP06] Cemal Yilmaz, Myra B Cohen, and Adam A Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 32(1):20–34, 2006. (cited on Page 7)

https://doi.org/10.24355/dbbs.084-202102191146-0

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Braunschweig, den 08. Januar 2021

https://doi.org/10.24355/dbbs.084-202102191146-0